MC68HC11 Laboratory Manual

N. Natarajan

Department of Electrical and Computer Engineering
University of Michigan-Dearborn
4901 Evergreen Road
Dearborn-48128

nnarasim@umich.edu

Contents

Chapter 1

Introduction to HC11

1.1 Objective

To become familiar with HC11 and using BUFFALO utilities, interacting using the terminal program, transferring files, editing-assembling-loading programs and executing them.

1.2 Tasks

1.2.1 Getting started with HC11

1. Disconnect power from the 68HC11
2. Connect 68HC11 to your computer using the serial cable that came with the machine.
3. Start the terminal program Hyperterm. The program is already configured to communicate with the 68HC11.
4. Power up 68HC11

You should see the following prompt

```
BUFFALO 2.5 (ext) - Bit User Fast Friendly Aid to Logical Operation
 >
```

Type `h` (for help) and press the enter key. You should see the following help screen

```
ASM [<addr>]  Line assembler/disassembler.
    /         Do same address.           ...
    CTRL-J    Do next address.           ...
```

```
    CTRL-A   Quit.
BF <addr1> <addr2> [<data>]  Block fill.
BR [-][<addr>]  Set up breakpoint table.
...
...
```

1.2.2 Looking at memory

To see what is stored in the HC11 memory, we use the memory dump command. The instruction for dumping memory is `MD`. Let us look at what is present in locations `E000` to `E3FF`. At the prompt, enter the command `MD E000 E3FF`. You should see:

```
>MD E000 E03F

E000 CE 10 0A 1F 00 01 03 7E B6 00 86 93 B7 10 39 86               9
E010 00 B7 10 24 8E 00 68 BD E3 40 CE 00 4A DF A7 86    $  h  @  J
E020 D0 97 A6 CC 3F 0D DD 69 BD E1 9A 7F 00 A9 7C 00     ?  i
E030 A9 7F 00 AB B6 10 3C 84 20 27 35 86 03 B7 98 00       <  '5
>
```

Each line of output consists of three parts. First there is the memory address, for example `E020`. This is followed by 16 bytes of data. These are contents of 16 locations starting from the address. For example, in the above example, memory location `E020` contains `D0`, location `E021` contains `97` etc. Recall that all numbers are in hex. After the 16 bytes come 16 characters. Each of the bytes is interpreted as an ASCII code and the character that the code represents is shown. Non printable and non-ascii characters are shown as a space.

Exercise

1. Look up the ascii code for the letter `J`. Can you locate it in the memory dump shown above?

2. Determine the contents of the locations `FFD0` to `FFFF`. Write down the contents in your lab notebook. You will need these values later in the course.

1.2.3 Modifying memory

Now that you know what is in the memory, let us try to modify the contents of the memory. Let us store the following values in locations starting from `C100`: `B6, D0, 00, BB`. To modify memory, we use *Memory modify* command, `MM`. When you

modify the memory, the command will first echo the value already in memory. If you don't want to change it, press the space bar, command will move on to the next memory location. If you want to change the value, just type the value (in HEX). If you make a mistake, **do not press the backspace key**. Continue typing. `MM` will only look at the last two characters you typed. So if you type `AB872B`, you have in effect typed `2B`. Once you have entered the correct value, press the space bar. When you are all done, press the enter key. Here is a quick list to remind you:

1. To leave memory unaltered but to move to next location, press SPACE bar
2. To modify memory and then move to next location, enter the data and then press the SPACE bar
3. When entering the value, only the last two characters are used. So if you make a mistake continue typing
4. When done, press the enter key

The following shows the interaction for entering the four values. Since `MM` echoes the previous values stored in memory, you may see different numbers:

```
>MM C100
C100 A7 B6 BB D0 C2 00 32 BB
>
```

After every memory modify command get into the habit of running the memory dump command to make sure that the memory was modified the way you wanted.

Exercise

1. Try modifying memory at location `E000`. What happens when you do it?
2. Modify the memory at three locations starting from `D000` to the following values `10 32 A8` and make sure that the changed did take place.
3. Turn the power to HC11 (not the PC!) off and then on again. What happened to the changes you made to locations `D000` to `D002`? Why?
4. Look at memory locations `E000` to `E00F`. Did any of them change when you cycled the power?

1.2.4 Writing and entering your first program: Using MM

Let us write a short program that will add three numbers stored in locations `D000`, `D001` and `D002`. This will be done in five steps

1. First we will load the register **A** with the value stored in location `D000`
2. Next we will **add to** the register **A** the value stored in location `D001`
3. Next we will **add to** the register **A** the value stored in location `D002`
4. Next we will store the value in the register **A** in location `D003`
5. We will return control back to BUFFALO.

We consult the little pink book and find that the command to load a value into register **A** is called `LDAA`. Since we want to load from memory, we need the `EXT`[1] mode of the command. The code for the command, commonly known as the operation code, or more simply OPCODE, is `B6`. So our first instruction is **B6 D0 00**. We say that the instruction is **LDAA D000**. Similarly, we find that the second instruction is **BB D0 01** which corresponds to **ADDA D001** The command for storing data is called `STAA` and the OPCODE is **B7**. The instruction to return control back to BUFFALO is **3F** and is called **SWI**. Thus the entire program is:

B6 D0 00 BB D0 01 BB D0 02 B7 D0 03 3F

Now that you have written the program, you have to load it into HC11 memory. You do that using the memory modify command. Before you can do that, you have to decide *where* you want to store the program. Check with your TA to see if he has a preferred location[2]. For now, we will use the locations starting at **C100**. So type the command **MM C100** and enter your program, one byte at a time.

1.2.5 Running your first command

First modify the locations `D000-D00f` to values you can easily recognize (but not all zeros). Assuming that you stored the program starting at location **C100**, you run the program by using the call command

[1] At this stage other modes will not make sense. You will soon learn about the other modes

[2] You can pretty much enter your program anywhere you have RAM. However, to make it easy for the TA when he goes from one student to the next, each TA may state certain standard locations

```
call C100
```

Note you can abbreviate any command by entering enough characters to identify it. Since no other command starts with the letter `c`, you could also have typed `c C100`[3]. When the control is returned to BUFFALO, it prints the contents of all the registers and you should see something like:

```
>CALL C100
P-C10C Y-AAAA X-AAAA A-EA B-AA C-D8 S-004A
>
```

Do a memory dump of locations `D000-D00F`. You should see something like:

```
>md D000 D00f
D000 10 32 A8 EA FF FF FF FF FF FF FF FF FF FF FF FF  2
>
```

Exercises

1. Modify the values in locations `D000-D00f` and run your program. Write down the values in the locations `D000-D003` and the value in register **A**, after you run the program.

2. Repeat the above 5 times and explain your results.

1.2.6 Entering your program: Using ASM

We will reenter your program, except we will store it at a different location. Enter the program starting at address **C200**, using the asm command. When you enter the ASM command, BUFFALO will display instruction currently stored in memory (or a question mark if it is not a valid instruction). You can press enter if you want to leave memory unchanged or type the command and then press the enter key. On power-up, the BUFFALO performs a memory check and fills all of RAM with **FF**. **FF** happens to be the code for `STX`. So ASM command will often display **STX $FFFF**[4]. Here is how I entered the program at location `C200`. What I typed is shown in **boldface**:

[3] An alternative to call is the go command, G. There is no difference between them if your program ends with `SWI`. However, if your code ends with an `RTS`, you have to use the call command.

[4] BUFFALO expects all its data to be entered in HEX. So you do **NOT** type the $ in front of numbers to indicate that the number is entered using HEX representation. However, BUFFALO adds the $ in front of the numbers written using HEX representation

```
>ASM C200
C200        STX $FFFF
            >LDAA D000
            B6 D0 00
C203        STX $FFFF
            >ADDA D001
            BB D0 01
C206        STX $FFFF
            >ADDA D002
            BB D0 02
C209        STX $FFFF
            >STAA D003
            B7 D0 03
C20C        STX $FFFF
            >SWI
            3F
C20D        STX $FFFF
            > CTRL-A
```

Note that to get out of the ASM command you need to type the control-A character.

Exerecises

1. Enter the program shown above and verify that the program is entered correctly.

2. Search through the memory starting at location `$E000` to find the string `User Fast Friend`. On my HC11 it is at location `E610`. Yours may be different. Locate the string and find out where the string is stored, i.e. the address of the first character. Modify the value `E610` in what follows with the address you found.

   Enter the following program starting at location `C300` and run it. What is the result of running your code?

   ```
   LDX #E610
   JSR FFC7
   SWI
   ```

3. (a) Modify the above program by changing E610 to D200.

(b) Find out the ascii codes for the letters in the phrase `THIS CLASS IS FUN`. To get you started, here are some codes: code for 'T' is 54, code for 'H' is 48 and code for I is 49.

(c) Enter the ASCII codes starting from location D200. After you enter the last ASCII code, enter the special code **04**.

(d) Run the program and write down what output you get.

(e) To see why you need the special code, do memory modify and change it to `0A`. Rerun the program and write down what you observe.

1.2.7 Entering your program: Using assembler

In this method, we will do all the work on the PC and eventually transfer the program to HC11. Start any text editor. **DO NOT USE A WORDPROCESSOR** such as Word, Wordpad etc. The best editor I am aware of the Programmer's file editor (PFE). This editor is part of the lab package that you can download from the web. Create a directory/folder where you will do all your work. In that directory, copy all the files from the lab pack.

Using the editor, create a file called `PROG1.ASM` and enter the following[5]:

```
*Name: your name goes here
*Uniqname: Your unique name goes here
*Class: ECE 373 (or your course number)
*Term: The term
*Date: Date you started the code
   ORG $C000 *this determines where the code will be stored
   LDAA $D000
   ADDA $D001
   ADDA $D002
   STAA $D003
   SWI
```

Save your file. Start a MSDOS. Change directory to where your files are stored. Execute the command

```
ASM PROG1
```

If there are no errors, you should see two new files, `PROG1.LST` and `PROG1.S19`. The first file for humans to read and is often called the LST file and the second is for the HC11 and is called the S19 file.

Go back to the BUFFALO prompt and type the command

[5]By tradition, assembly language programs are written in UPPERCASE letters, except if you are writing code for Unix and its derivatives.

```
LOAD T
```

In Hyperterm, use the ASCII file transfer command (use the Transfer menu item to get to it) to transfer the `PROG1.S19` file. When the transfer is complete, you should see the program in locations starting from C000.

Exercises:

1. Write the above code, assemble it (i.e. create the S19 file), transfer the S19 file to HC11 and run the program. Verify that the program works correctly.

2. Copy the file LAB1.ASM. The file has three errors in it. Assemble it and look at the error messages. Fix the errors and submit a corrected version.

Chapter 2

Introduction to Looping

2.1 Objective

To become familiar with elementary loops and simple input/outputs functions.

2.2 Simple Input/Output

One of the basic functionality provided by any operating system is input/output routines. BUFFALO provides several useful functions for performing input/output. In this lab, we will look at two output functions provided by BUFFALO. Unlike in high level languages, functions in machine language are known by their addresses. The two functions we will be using are in ROM at locations `$FFB8` and `$FFBB`[1]. Rather than use these hard to remember and hard to recognize numbers, it is customary to give them meaningful names. Unless you have a good reason to do otherwise, it is best to use the name suggested by the vendor, in this case Motorola. The 'official' names for these functions are `OUTA` and `OUT1BYT` respectively. In assembly language, we make the connection between a name (technically known as a label) and a value using the EQU command as shown

```
OUTA        EQU     $FFB8
OUT1BYT     EQU     $FFBB
```

NOTE: Labels should be written starting from column 1. If a line does not have a label, it should start with a space or a tab or a comment character.

[1]We will indicate HEX values with the prefix $. However, data that you would be entering in BUFFALO, as part of memory modify or register modify will be shown without the prefix $ although it will be understood that the numbers are written in HEX.

2.2.1 The function `OUTA`

The function `OUTA` will transmit whatever is in register A over the serial communication line that is connected to the PC. What the PC does with this value depends on the terminal program that is used to communicate with the HC11. Under normal circumstances, the terminal program will interpret the value as an ASCII code and display the corresponding character on the screen.

Exercise: Power up HC11 and at the BUFFALO prompt try the following and write down what you see.

1. Issue the command RM (for register modify) and press the space bar till you see the **A** register. Enter the value 31 and press enter as shown below:

```
 >rm
P-AAAA Y-AAAA X-AAAA A-AA B-AA C-D0 S-004A
 P-AAAA
 Y-AAAA
 X-AAAA
 A-AA 31

 >
```

   Now execute the command

```
 CALL FFB8
```

2. Repeat with the **A** register modified with the following values: `32, 33, 34, 21, 22, 23, 24, 25, 41, 42 43 44`

If you are using the simulator, turn on the log feature (by pressing both the shift keys). If you are working with a real HC11, you can copy and paste the contents of the terminal screen.

2.2.2 The function `OUT1BYT`

This is a more involved output function. To start with, the value to be printed must be in memory. If the value is in a register you will have to store it in memory first. Next, the value in the **X** register should be the address where the value is stored. Thus this function should be told 'where' and not 'what'. When you call this function, the function will send *two* characters to the PC. If the terminal program interprets these two characters as ASCII codes and displays the corresponding two characters, then the display would be the value written in HEX. *In addition, the function will increment the value in the* **X** *register.* This is useful when displaying a series of memory locations.

Exercise:

1. Using the MM command (memory modify) enter the following values in memory locations `D000, D001`, $\cdots$: `30 31 32 33 41 42 43 44`. Verify the values using the memory dump, MD, command.

2. Using RM, the register modify command, change the value in the **X** register to `D000`

3. Execute the command `CALL FFBB` and write down what the output was and also the value in the **X** register after the command is executed.

4. Repeat the the command `CALL FFBB` and write down the output and the new value in the **X** register.

5. Repeat the last part until you have performed 7 calls to `$FFBB`.

2.3 Branching

Conditional branching in HC11 is controlled by the state of one or more hardware flags. The state of a flag depends on the most recently executed instruction that affects the flag. Thus, if your branching depends on the result of some instruction, then it is your responsibility to make sure that none of the instructions between the instruction you are interested in and the branching instruction affects the flags that control the branching instruction. Thus, it is a good idea to follow the instruction that sets the flag by the branching instruction. In this lab, we will use the following conditional branch instructions:

BEQ	Branch if the Z flag is set
BNE	Branch if the Z flag is not set

Now, the `Z` flag is set after most instructions if the result of the instruction is a *zero*; or else it is cleared, i.e. not set. The two most important instructions that are often used to set/clear the flag are

CMP	Perform a subtraction and discard the answer. However, set the flags.
TST	Same as CMP except subtract the number zero

Thus, after the `CMP` command, the `Z` flag would be set if the two values that are compared are equal. Similarly, the `TST` command will compare a value (memory or register) with zero and set the Z flag if the value is zero.

Exercise:

1. Using the HC11 reference book, identify 5 instructions that do *not* affect the **C** flag, but affects some other flag.

2. Using the HC11 reference book, can you identify any instruction that does *not* affect the **V** flag, but affects some other flag?

3. Using the HC11 reference book, identify 5 instructions that always clears the **C** flag.

4. Using the HC11 reference book, identify an instruction that always sets the **C** flag.

2.4 Looping

2.4.1 Counting loops

This is by far the simplest and most used loop structure. In a counting loop, we perform a specific operation a given number of times. A counter controls how many times the loop is executed. The counter could be stored in memory or kept in a register. If it is kept in a register, it is your responsibility to make sure that the register is not inadvertently changed either by your code or by some third party function that you call. If you decide to use a register, your best bet is to use either the **B** register or **Y** register. The structure of the loop is as follows:

1: Initialize the counter to the number of times the loop is to be performed

2: Perform any other initializations

3: Test if the counter is zero. If so quit the loop

4: Perform the desired task

5: Perform any re-initializations

6: Decrement the counter

7: Go back to 3

8: Come here when you quit the loop

Note that there are two places where the code jumps to. One to (3) and the other to (8). When writing the code in assembly language, we would need two labels. In the code that follows, I have used the labels `FOO` and `BAR`.

Exercises:

1. Assemble the following code in your PC, transfer the S19 file to HC11, run the program and write down the output of the program.

```
;Name:
;email:
;date:
;
OUTA          EQU     $FFB8

              ORG $C100
              LDAB    #$9 ; USING REGISTER B AS A COUNTER
              LDAA    #$31 ; OTHER INITIALIZATION
FOO           TSTB        ; SUBTRACT ZERO FROM B
              BEQ BAR     ; QUIT IF THE Z FLAG IS SET, I.E. B=0
              JSR OUTA    ; DO THE TASK
              INCA        ; RE-INITIALIZE
              DECB        ; DECREMENT THE COUNTER
              BRA  FOO    ; GO BACK

BAR
              SWI
```

2. Modify the above program so that the program prints the upper case letters `A` to `Z`. Clearly indicate the changes you made.

3. The following function is equivalent to the function shown above[2].

```
; continued from the previous function

              ORG $C200
              LDAB    #$9 ; USING REGISTER B AS A COUNTER
              LDAA    #$31 ; OTHER INITIALIZATION
JUBJUB
              JSR OUTA    ; DO THE TASK
              INCA        ; RE-INITIALIZE
              DECB        ; DECREMENT THE COUNTER
              BNE  JUBJUB    ; GO BACK
```

[2]You can have as many functions as you want in the same file. Make sure that when you change the ORG, the functions do not overlap. You can determine this by looking at the LST file

```
        SWI
```

Verify that `CALL C100` and `CALL C200` produces the same output. Explain why this is so, and how and why the loop terminates.

2.4.2 One, two! One, two! And through and through ... Marching through memory

Often loops are combined with marching through memory and operating on consecutive memory location. In this case, the **X** (and/or **Y**) register is initialized to a starting memory address. Inside the loop, memory is accessed using `IND,X` addressing mode. This operates on memory whose address is computed using the value in **X** register. At the bottom of the loop, **X** is incremented, so that next time around the loop, the operation is performed on the next memory location[3]. The following program shows prints using `OUTA` the values stored in 9 consecutive locations starting from location `$D000`. Note we are using FCB which is the assembly language equivalent of memory modify.

```
        ORG     $C300

        LDAB    #$09
        LDX     #$D000 *Don't forget the #

VORPAL  TSTB
        BEQ     SWORD

        LDAA    0,X
        JSR     OUTA

        INX
        DECB
        BRA     VORPAL

SWORD   SWI

; FOLLOWING SAME AS MM D000 FOLLOWED BY: 55 6F 66 4D 2D 44 62 72 6E
```

[3]In some cases the memory has to be accessed in the reverse order. In this case, **X** starts at the end, and at the bottom of the loop, **X** is decremented.

```
              ORG $D000
              FCB  $55, $6F, $66, $4D, $2D, $44, $62, $72, $6E
```

Run the above program using `CALL C300`[4]. Modify the above program as shown below and explain what the program does. Do you see why we do the `DEX` before we access memory?

```
              ORG    $C400

              LDAB   #$09
              LDX    #$D000
              ABX

SNICKER       TSTB
              BEQ    SNACK

              DEX

              LDAA   0,X
              JSR    OUTA

              DECB
              BRA    SNICKER

SNACK         SWI

              ORG $D000
              FCB  $55, $6F, $66, $4D, $2D, $44, $62, $72, $6E
```

Now for some other useful examples. Explain what each of them does. Run the programs and using memory dump, verify that your explanation is correct. Each of the functions start with an `ORG` command.

```
              ORG   $C500
              LDAB  #$10
              LDX   #$D000

CALLOOH       TSTB
              BEQ   CALLAY
```

[4]Don't forget to assemble it and then transfer the S19 file to the HC11!

```
                LDAA  0,X
                ADDA  $10,X
                STAA  $20,X

                INX
                DECB
                BRA CALLOOH
CALLAY
            SWI

            ORG $D000
            FCB $55, $6F, $66, $4D, $44, $62, $72, $6E
            FCB $41, $42, $43, $44, $45, $46, $47, $48
            FCB 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

; NOTE: YOU CAN ENTER NUMBERS EITHER IN DECIMAL OR HEX!
; AFTER YOU RUN THE PROGRAM, DO
; MD D000 D02F
; TO SEE WHAT THE PROGRAM DOES
```

The following program prints the sixteen values stored in the consecutive location starting from $D020. To make the output more user friendly, each number is followed by a comma and a space.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ORG     $C600
                LDAA    #$0A
                JSR     OUTA
                JSR     OUTA

                LDAB    #$10
                LDX     #$D020

KINGS           TSTB
                BEQ     CABBAGES

                JSR     OUT1BYT
                LDAA    #$2C
                JSR     OUTA
                LDAA    #$20
                JSR     OUTA
```

```
                DECB
                BRA     KINGS
CABBAGES
                LDAA    #$0A
                JSR     OUTA
                JSR     OUTA

                SWI
```

2.5 Other Conditional Branches

2.5.1 Signed and unsigned numbers

So far we have only considered comparing two values and branching if they are equal (`BEQ`) or not equal (`BNE`). Designers of HC11 have provided several other branches. When comparing two numbers, it is important to distinguish between signed and unsigned quantities. Note that the internal hardware does not deal with numbers *per se* bit with binary patterns which are interpreted as numbers. Conversely, numbers are represented by bit patterns and it up to you to chose the appropriate interpretation. For example, using binary representations, an 8-bit quantity can represent any number from 0 to $2^8 - 1$ or 0 to 255. This is the unsigned representation. What happens if we use an 8-bit counter and starting at zero and incrementing it by one? The value will go from one to two to three etc. all the way to 255 and then roll over to zero and start all over again. This is the *unsigned* interpretation. Now suppose you start at zero and decrement by one. The value will go from zero to 255 to 254 etc. But starting from zero decrementing by one you get **-1**. Thus 255 really represents **-1**. This is the signed interpretation. [5] This dual interpretation is quite natural and does not pose any problems except when performing comparisons. In the signed interpretations, numbers 0 to 127 are non-negative, and numbers 128 to 255 are negative, with 128 representing **-128**, 129 representing **-127** ... and 255 representing **-1**. Note that the negative numbers come **after** the positive numbers thus when comparing two numbers of the **opposite** sign, the sense of inequality are reversed. Keeping this in mind, HC11 hardware designers have two sets of branch conditions, one for signed numbers (where special attention is given if the two numbers that are

[5] This dual interpretation is something I use often. It is easy to think of the clock, and say the minute setting. Starting at zero and incrementing by one, I get one, two, $\cdots$ to 59 and back to zero. When I set the alarm on this clock I cannot go back, only forward. So, if I want to decrease the minute value by 5, I have to advance it by 55. Thus adding 55 is same as subtracting 5, or in terms of arithmetic 55 is has the dual interpretation of -5.

compared are of opposite sign) and one for unsigned comparison where no special cases need be considered.

2.5.2 compare and branch instructions: Unsigned

The following is based on the section named **Branches** in the *M68HC11E Series Programming Reference Guide* commonly referred to as the little pink book. In each case, the instruction compares a value in register r with a value in memory M.

Branch to `LOC`if A > MEM

```
CMPA MEM
BHI LOC   *Branch if HIgher
```

Branch to `LOC`if A <= MEM

```
CMPA MEM
BLS LOC   *Branch if Lower or Same
```

Branch to `LOC`if A >= MEM

```
CMPA MEM
BHS LOC   *Branch if Higher or Same
```

Branch to `LOC`if A < MEM

```
CMPA MEM
BLO LOC   *Branch if Lower
```

2.5.3 compare and branch instructions: Signed

Branch to `LOC`if A > MEM

```
CMPA MEM
BGT LOC   *Branch if Greater Than
```

Branch to `LOC`if A <= MEM

```
CMPA MEM
BLE LOC   *Branch if Lesser or Equal
```

Branch to LOCif A >= MEM

```
CMPA MEM
BGE LOC   *Branch if Greater or Equal
```

Branch to LOCif A < MEM

```
CMPA MEM
BLT LOC   *Branch if Less Than
```

2.5.4 An example: HEX2BCD

Consider the problem of converting a number represented in binary to its equivalent representation in BCD. Without going into too much detail, the rule for numbers between 0 and 99 (two digit numbers) is as follows:

1. For numbers between 0 and 9 (inclusive), no change.
2. For numbers between 10 and 19 (inclusive), add 6.
3. For numbers between 20 and 29 (inclusive), add 12.
4. ··· ···
5. For numbers between 90 and 99 (inclusive), add 54.

Without using Loops

Here is a program that takes the value in location $D000, convert it to BCD and store it in location $D001. First a program that does not use loops. Before running the program use MM to change the value in location $D000. Recall MM requires you to enter the value in HEX. So make sure the value you enter is between 00 and 63.

```
;;;;;;;;;;;;;;;;;;;;;;;; START OF HEX2BCD ;;;;;;;;;;;;;;;
;
; THIS PROGRAM TAKES THE VALUE IN LOCATION $D000 AND
; CONVERTS IT TO ITS BCD REPRESENTATION AND STORES THE
; RESULT IN LOCATION $D001
;
; THE NUMBER IN LOCATION $D000 SHOULD BE BETWEEN 0 AND 99 ($63)
; NUMBERS BIGGER THAN 99 ARE IGNORED AND LEFT UNCHANGED.

; THE PROGRAM ADDS MULTIPLES OF 6 AS APPROPRIATE
```

```
; IT USES CODE REUSE IN THIS SIMPLE FORM. TO ADD 54, IT
; IT ADDS 6 AND THEN FALLS DOWN TO THE CASE OF ADDING 48.
; TO ADD 48, IT ADDS 6 AND THEN FALLS DOWN TO ADDING 42, ETC.

    ORG $C000
    LDAA  $D000

    CMPA #10
    BLO  DONE

    CMPA #20
    BLO ADD6

    CMPA #30
    BLO ADD12

    CMPA #40
    BLO ADD18

    CMPA #50
    BLO ADD24

    CMPA #60
    BLO ADD30

    CMPA #70
    BLO ADD36

    CMPA #80
    BLO ADD42


    CMPA #90
    BLO ADD48


    CMPA #100
    BLO ADD54

    BRA DONE *NUMBER TOO BIG. LEAVE IT ALONE
```

```
ADD54
    ADDA #6
ADD48
    ADDA #6
ADD42
    ADDA #6
ADD36
    ADDA #6
ADD30
    ADDA #6
ADD24
    ADDA #6
ADD18
    ADDA #6
ADD12
    ADDA #6
ADD6
    ADDA #6

DONE
    STAA $D001

; NOW PRINT THE VALUES
    LDX #$D000
    JSR OUT1BSP *SAME AS OUT1BYT EXCEPT PRINTS A SPACE AFTER THE NUMBER
    JSR OUT1BYT

    SWI

;;;;;;;;;;;;;;;;;;;;;;;; END OF HEX2BCD ;;;;;;;;;;;;;;;
```

Here are some sample runs. Note that the value $23 is converted to $35.the value $29 is converted to $41.

```
>MM D000

D000 FF 23

>C C000
23 35
P-C04B Y-AAAA X-D002 A-20 B-AA C-D9 S-004A
```

```
>MM D000

D000 23 29

>C C000
29 41
P-C04B Y-AAAA X-D002 A-20 B-AA C-D9 S-004A
>
```

Using Loops

Here is a version using loops. We first copy A to B. Then in a loop check if B is greater than or equal to 10. If so, we subtract 10 from B and add 6 to A. If not, we are done.

```
;;;;;;;;;;;;;;;;;;;;;;;; START OF HEX2BCD ;;;;;;;;;;;;;;;
;
; THIS PROGRAM TAKES THE VALUE IN LOCATION $D000 AND
; CONVERTS IT TO ITS BCD REPRESENTATION AND STORES THE
; RESULT IN LOCATION $D001
;
; THE NUMBER IN LOCATION $D000 SHOULD BE BETWEEN 0 AND 99 ($63)
; NUMBERS BIGGER THAN 99 ARE IGNORED AND LEFT UNCHANGED.

; THE PROGRAM USES LOOPS AND REPEATED SUBTRACTION


OUT1BYT     EQU $FFBB
OUT1BSP     EQU $FFBE

    ORG $C000
    LDAA  $D000

    TAB          *COPY A TO B
LT
    CMPB #10     *IS B < 10
    BLO  DONE    *IF SO WE ARE DONE

    SUBB #10     *B <- B-10
    ADDA #6      *A <- A+6
    BRA LT
```

```
DONE
    STAA $D001

    LDX #$D000
    JSR OUT1BSP *SAME AS OUT1BYT EXCEPT PRINTS A SPACE AFTER THE NUMBER
    JSR OUT1BYT

    SWI

;;;;;;;;;;;;;;;;;;;;;;;; END OF HEX2BCD ;;;;;;;;;;;;;;;
```

Chapter 3

Functions and bit manipulations

3.1 Objective

To become familiar with bit level operations and writing functions. This lab also illustrates the use of random numbers for testing functions. Bit level operations are used to control light emitting diodes connected to PORTA as well as for monitoring external circuitry.

3.2 What you should do

You will be writing several functions in this laboratory exercise. You should have only one file and you should add the new function at the end of your older functions. Also, you will be adding items to the data section. You should **not** delete earlier data items. Your main code will be changing. You should insert your new main code before the older one, so that the most recent main code will start immediately after the `ORG` statement. Try not to delete any code from your file.

3.3 String outputs

In the last lab, we saw how to write a single byte as an ascii character. To write a string, it is tedious to load A with one character at a time and then calling `OUTA` each time. A better approach is to put all the characters in consecutive memory locations and print them all in a loop. To do this, we need two pieces of information, where to start and where to end. The common approach to such situation is to specify where to start and use a special value, known as sentinel, to indicate the end. Some of you may have used special values such as zero, one, or 9999. It is entirely up to the programmer, but for character strings, the three

most often used sentinels are zero (also known as ASCII-Z string), 26 (also known as CONTROL-Z string, or old DOS string), 4 (EOT string).

The programmers of BUFFALO use EOT string and you have one of two choices: rewrite BUFFALO routine and use some other sentinel, or use 4 as the sentinel and remember to place it after each string. The rest of the lab assumes that you will use the EOT string. Two functions that BUFFALO provides for printing strings are `OUTSTRG` at location `$FFC7` and `OUTSTRGO` at location `$FFCA`. The difference between them is that the former will print the string on a new line, while the latter will continue the string from wherever the cursor happens to be. Both these functions must be told where the string is. You do this by loading the **X** register with the starting address where the string is stored before calling the function. To enter strings in memory, we use FCC directive. The label associated with the FCC will be automatically `EQU`ated to the starting address of the string. Type the following code and verify that `OUTSTRG` does indeed print a string.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
OUTSTRG     EQU $FFC7
OUTSTRGO    EQU $FFCA

; program section. set origin to $C100
        ORG $C100
        LDX #ABOUTME    *STARTING ADDRESS OF THE STRING
        JSR OUTSTRG
        SWI

; data section. set origin to $D000
        ORG $D000
ABOUTME FCC /Hello, my name is ===your name =====/
        FCB 4 ;dont forget this
;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.4 Writing your first function

Before you write your first function, you should observe some standard conventions. Your programs should always start at location $C100, or as specified by your TA. Your data should always start at location $D000 or as specified by your TA. A small amount of data can be stored starting at location $0000 (Page 0), but you do not have too much space, 30 bytes or so. It is a matter of taste whether you write the data section first or the program section first. You can not mix and match. I personally prefer the following order: Page 0 section first, data section

next and then the program section, though it is easier to follow the code if the program section precedes the data section as in the previous example.

The program section should start with the main code, i.e. the code you want to execute. The main code should be followed by various functions. The order is not important.

What is a function? A function, also known as a *subroutine* is a *self contained* code that implements a well defined functionality. What do I mean by self contained? You should be able to draw a line above and below your code for the function, and make sure that

1. Only way to branch **out of** the two lines is with `JSR` or `BSR` or `RTS` instructions. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect

2. Only way to branch **into** an instruction between the two lines from an instruction outside the two lines is with a `JSR` or `BSR` instruction. If you find any other branching instruction such as `BRA`, `BEQ` etc. then your code is most likely incorrect.

The function should terminate with a return from subroutine `RTS` instruction. It is a good programming practice not to have more than one `RTS` statement in any function.

Once you have written your function, it is there for you to use as many times as you need. To use the function, you should know where its first instruction is located in memory (technically known as the entry point). If you use the assembler to create the function, you can place a label before the very first instruction. The assembler will automatically `EQU`ate the label with the first instruction. If the function needs any additional information, they will have to be supplied by the user prior to using the function (technically known as binding). The function you will be writing will use one of the registers for binding. Also, many functions will return some useful value to the caller. In this case, it is a good idea to return the value in one of the registers.

Unless you write functions that do absolutely nothing (technically known as stubs), the function will use and modify one or more registers. If the caller happens to keep valuable data in one of these registers, then you have a potential problem. There are two possible solutions: The caller could save the values in the registers before calling your function, and then restore it after your function returns. Alternatively, your function can save the values in the registers it uses and then restore the values before it returns. The first approach more efficient but the second approach will result in fewer bugs. I strongly recommend that you get into the habit of writing functions that clean up after themselves and restore

the registers the way they were before the functions used them[1].

Here are the basic rules for writing functions

1. Decide on its functionality. Don't try to create a Swiss army knife that has multiple functionalities built in. Your function must do only one thing, and it must do it well. Your documentation for the function must clearly state the functionality

2. Decide on its name. Pick a meaningful name but keep the name to 8 characters or less

3. Decide on the registers that will be used to pass information **to** the function. 8-bit values can be sent using **A** or **B** registers. 16-bit values can be sent using **X** or **Y** registers[2].

4. Decide what registers will be used to **return** values back to the caller. 8-bit values can be returned using **A** or **B** registers. 16-bit values can be returned using **X** or **Y** registers.

5. Decide what registers the function will use and which of these will be restored back at the end. Clearly document which registers will be used and **not** restored back as the registers that are modified by the function.

6. Write the function. Avoid the temptation of writing the function first and then worrying about the other items!

As an example, we will write a simple function called `RAND` that will return a random value every time it is called. How does this function work? The function starts with a seed. We use an 8-bit number as a seed. The seed is used to calculate a random number using some formula. To make sure we get a different number every time the function is called, the seed value is changed. Typically, the random number that is generated is used as the seed for the next random number. Thus, we are actually generating a random sequence starting with some initial seed. The formula[3] it uses is simple: It shifts the seed value left, and adds with carry the value 20. The function needs one byte of storage to keep track of the seed. This storage will be allocated in the data section using the `FCB` directive as

```
SEED    FCB    0
```

[1]The only exception is the register that is used to return a value to the user. Clearly, these registers should not be restored to their original value!

[2]For example, `OUTA` expects the data in the**A** register, while `OUT1BYT` expects the information in the **X** register.

[3]I use this as a quick and dirty 8 bit random number generator. It is not the best, but the code is only 5 line long!

You use FCB to initialize consecutive memory loication (when you transfer the code from the PC). The label associated with the instruction is needed to determine the address where the instruction forms the constant. The assembler will automatically EQUate the label with the address associated with the FCB directive.

Type the following code, assemble it, transfer the S19 file to the HC11, and test your program by CALL $C100. Repeat[4] the CALL statement and verify that the value in the **A** changes after each call.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
    ORG $C100
    JSR RAND
    SWI


;;;;;;;;;;;;;;;;;;;;;;Start: RAND ;;;;;;;;;;;;;;;;;
; Function: RAND
; Purpose: Generate a random number
;
; Inputs: None
; Outputs: A random value returned in A registers
;
; Registers modified: A register (which has a random value)
;
; Memory usage: The most recently generated random number is
;                stored in memory with label LSTRAND
;                This value is used to generate the next value
;
; Notes: Not the best random number generator around but does a
;        halfway decent job.
;
;        works as follows:
;        shift the last random value left and add 20 with carry
;
;



RAND        LDAA    SEED
        LSLA
```

[4]In BUFFALO, if you press the enter key at the prompt, BUFFALO will repeat the last instruction. So you don't have to type the CALL every time. Just press the enter key.

```
        ADCA    #20
        STAA    SEED ;don't forget to save it back
        RTS
;;;;;;;;;;;;;;;;;;;;;;;;End: RAND ;;;;;;;;;;;;;;;;;;;

;
; DATA SECTION
;

    ORG $D000
SEED     FCB     0

;;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.4.1 On random sequences

When you are done running the program a few times, you reload the s19 file and try again. *You will get the same sequence all over again!* Now this is useful when you are debugging programs but is absolutely useless if you are writing a game program. *Every game will be 100% predictable!* The way to overcome this is to change the seed everytime you start your program in some unpredictable way. One simple solution is to use the low order byte of the clock inside the HC11. This clock is in locations `$100E-$100F`. So you can use the second location `$100F` as the initial seed.

3.5 Your second function

For your second function, you will be writing a function that will print an 8-bit value in HEX first and then in binary. We will call this function `PRBINARY`

An 8-bit number requires 2 hex-digits to print and BUFFALO has two routines to help you, one to print the left digit and the other to print the right digit. These are called `OUTLHLF` for out-left-half and `OUTRHLF` for out-right-half respectively.

To print it in binary, we will use the shift left instruction `LSL` . This instruction will shift all bits left by 1 place and the (left most) bit that is shifted out will be stored in the carry flag. I.e. if an 8-bit value before shifting was `abcdefgh` then the value after shifting will be `bcdefgh0`. Here each of the letters `a` to `h` represent bits. The carry flag will be set to `a`. Thus the value to be printed will be shifted left 8 times. After each shift, the A register will be loaded with either the ascii

code for `0` or the code for `1` depending on whether the carry is cleared or set[5]. Note the function involves a counting loop.

We now have to decide register usage: We will use the **A** register to pass the value to the function. Internally, this value will be moved to **B** as **A** is needed in all the calls to BUFFALO routines. We need a counter, and we will use the **X** register to keep count. As a good programming practice, we will store and restore all registers we will use.

Here is the complete code for the function.

```
;;;;;;;;;;;;;;;;;;;;Start: PRBINARY;;;;;;;;;;;;;;;;;;;
; Function: PRBINARY
; Purpose: To print a value in binary
;
; Inputs: Value to be printed is passed in the A register
;
; Returns: None
;
; Registers affected: None. The values in the registers are stored
;            first and these values are restored at the end.
;
; Notes: The output consists of two parts. The value in A is first
;        printed in HEX, and then in binary
;

PRBINARY

; first save the registers we will be using: a, b and x
    PSHA
    PSHB
    PSHX

; copy a to be for later use

       TAB

; print a as hex number (2 digits).

; print the left digit
    JSR OUTLHLF
```

[5]Conveniently, the code for `1` is one more than the code for `0`. So we load **A** with the code for `0` and add the carry to it

```
; the function destroys the value in a,
; so re load it! then print second digit
    TBA
    JSR OUTRHLF

; now print a colon and some spaces
    LDAA #':'
    JSR OUTA

    LDAA #' '
    JSR OUTA
    JSR OUTA
    JSR OUTA

; now print it in binary
; b has the value to be printed (recall the old tab)
;
; shift b  to the left by one bit and print '0' or '1' depending
; on what is in the carry flag
; repeat 8 times.
;
; we will use X register as counter
; to print what is in carry flag, we will load A
; with the code for '0' ; and add the carry to the code
; prior to calling OUTA

    LDX #8 *COUNTER

PRBLOOP CPX #0
    BEQ PRBDONE

    LDAA #'0'
    LSLB
    ADCA #0
    JSR OUTA
    DEX
    BRA PRBLOOP
PRBDONE
    JSR OUTCRLF
```

```
; restore the registers
    PULX
    PULB
    PULA
    RTS

;;;;;;;;;;;;;;;;;;;;;;End: PRBINARY;;;;;;;;;;;;;;;;;;
```

3.5.1 Test your function

We can test the function by loading different values in the A register. The random number generator we wrote first comes in useful here! Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

UCASE       EQU $FFA0
WCHEK       EQU $FFA3
DCHEK       EQU $FFA6
INIT        EQU $FFA9
INPUT       EQU $FFAC
OUTPUT      EQU $FFAF
OUTLHLF     EQU $FFB2
OUTRHLF     EQU $FFB5
OUTA        EQU $FFB8
OUT1BYT     EQU $FFBB
OUT1BSP     EQU $FFBE
OUT2BSP     EQU $FFC1
OUTCRLF     EQU $FFC4
OUTSTRG     EQU $FFC7
OUTSTRGO    EQU $FFCA
INCHAR      EQU $FFCD
VECINIT     EQU $FFD0

    ORG $C100
    LDX  #ABOUTME
    JSR  OUTSTRG
```

```
    JSR  RAND
    JSR  PRBINARY
    SWI
;;; INSERT YOUR CODE FOR PRBINARY HERE
;;; INSERT YOUR CODE FOR RAND HERE

    ORG $D000
;;; INSERT ALL YOUR DATA (FCB, FCC, RMB etc.) here
ABOUTME FCC / INFORMATION ABOUT YOU /
    FCB 4
SEED     FCB     0

;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.6 Setting bits

We will now write a function that will set a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will set bit #4 in memory location $00. Recall that the bits are numbered right to left starting with bit #0. To set a bit, we use the `ORA` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here

    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00
    JSR     PRBINARY *PRINT BEFORE
    JSR     SETBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER
    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: SETBIT4
; Purpose: SETS bit #4 in memory location $00
; Registers modified none
;
```

```
SETBIT4
    PSHA

    LDAA $00
    ORAA #%00010000
    STAA $00

    PULA
    RTS


;; INSERT THE CODE FOR FUNCTIONS RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.
;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.7 Clearing bits

We will now write a function that will clear a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will clear bit #4 in memory location $00. To clear a bit, we use the `AND` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here

    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    JSR     SETBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER SET
```

```
    JSR     CLRBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER CLEAR

    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: CLRBIT4
; Purpose: Clears bit #4 in memory location $00
; Registers modified none
;

CLRBIT4
    PSHA

    LDAA $00
    ANDA #%11101111
    STAA $00

    PULA
    RTS


;; INSERT THE CODE FOR FUNCTIONS SETBIT4 RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
;;;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;;
```

3.8 Toggling bits

We will now write a function that will toggle (make it zero if it was a one; make it one if it was a zero) a particular bit in some memory location. The function should modify the value in the memory in such a way that it only affects the specific bit without changing any other bit. For definiteness, we will toggle bit #4 in memory location $00. To toggle a bit, we use the `eor` instruction. Write the following program, and test it by repeating the call to $C100 from the BUFFALO prompt.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here
```

```
    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 1 TOGGLE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 2 TOGGLE

    JSR     TGLBIT4
    LDAA    $00
    JSR     PRBINARY *PRINT AFTER 3 TOGGLES

    SWI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: TGLBIT4
; Purpose: tOGGLES bit #4 in memory location $00
; Registers modified none
;

TGLBIT4
    PSHA

    LDAA $00
    EORA #%00010000
    STAA $00

    PULA
    RTS


;; INSERT THE CODE FOR  CLRBIT4 SETBIT4 RAND AND PRBINARY HERE
```

```
    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.9 Testing bits

Often we have to take a decision based on whether a bit is set or not in memory. To test if one or more bits are set, we clear all other bits and see if the result is zero. If so, none of these bits are set. If not, at least one of them was set. The following program will print `YES` if bit #4 in memory location $00 is set. Or else it will print `NO`.

```
;;;;;;;;;;;;;;;;;;;;;;start of code ;;;;;;;;;;;;;;;;;;;;;;
;; Insert standard buffalo equates here
BIT4 EQU %00010000 ; THIS EQUATE MAKES THE CODE MORE READABLE
    ORG $C100
    JSR     OUTCRLF *NEED THIS FOR OUTPUTS TO LINE UP!
    JSR     RAND
    STAA    $00

    JSR     PRBINARY *PRINT BEFORE

    LDAA    $00
    ANDA    #BIT4
    BEQ     NOPE
    LDX     #YESSTR
    JSR     OUTSTRG
    SWI

NOPE
    LDX     #NOSTR
    JSR     OUTSTRG
    SWI


;; INSERT THE CODE FOR  CLRBIT4 SETBIT4 RAND AND PRBINARY HERE

    ORG $D000
;; ALL THE DATA ITEMS GO HERE.$
YESSTR  FCC /YES/
        FCB 4
```

```
NOSTR   FCC /NO/
        FCB 4
;;;;;;;;;;;;;;;;;;;;;;end of code ;;;;;;;;;;;;;;;;;;;;;;
```

3.10 Hardware Interfacing

Working with random numbers is fine, but we want to do something useful. In HC11, there area special memory locations called `PORTS`. The special nature of these locations allows us to have direct access to the individual bits using external circuitry. Each bit in the port has an I/O line associated with it. This line provides access to the bit. The bits in the port can be one of two types:

1. A bit in a port could be an **input** bit. If this is the case, then we can only set or clear the bit using external electrical circuit connected to the I/O line associated with the bit. *This means that the code you wrote earlier to set or clear a bit will have no effect on the bit.* You can however check the bit and take appropriate action. To set the bit, you have to set the voltage of the I/O line above 3 volts (without exceeding the supply voltage of 5 volts). To clear the bit, you have to set the voltage of the bit to below 2 volts (without going below zero).

2. A particular bit can be an **output** bit. If this is the case, we can set or clear the bit in our program and the bit will control the I/O line associated with the bit. If the bit get set, then the voltage on the line will go to 5 volts. If the bit is cleared, the voltage on the line will go to zero. **Make absolutely sure that you do not connect any external device that can control the voltage (such as a voltage source) to the line.** The single biggest reason why HC11 ports get burnt is when some external device tries to send the voltage on the line to zero while your program tries to send it to 5 volts or vice versa. **If you are concerned about damaging the port. always connect 2.2K or larger resistor in series with the port. This will limit the port current to 1 mA or less.**

3.10.1 `PORTA` at location `$1000`

In this experiment we shall work with `PORTA` which is at memory location `$1000`. The bits of `PORTA` are designated as `PA7, PA6, PA5, ··· PA0`. If you are using the `FOX11` board, the lines associated with these ports are clearly labelled. If you are using `CMD11E1` from Axiom, the lines are the first 8 pins in the `MCU CONNECTOR`. The bits `PA0, PA1, PA2` are inputs. This means you can connect external circuits to these pins. The bits `PA3, PA4, PA5, PA6` are outputs. This means you can

drive devices from these pins (as long as you do not supply more than 5 mA). `PA7` is bidirectional and you, as the programmer, can configure the pin as either input or output.

WARNING: You may have wired the port as input and either by accident or oversight, may set the pin as an output pin. This can seriously damage the pin if the pin carries currents in excess of 5 mA or so. To prevent the damage, make sure there is a current limiting resistor (4.7K) in series with the pin. This warning is applicable to any pin that is bidirectional.

WARNING: If you lend your HC11 to anyone, make sure you disconnect any circuit that may be connected to `PA7`.

Special instructions for `CMD11E1` users

1. Locate the jumper `JP13` and make sure that it is open.

2. Locate the MCU port. This is a dual row 34-pin Berg style connector. Locate pin #1 on the port. This is identified with the number 1 on the front of the board. On the other side (solder side), pin #1 is identified by a square solder.

3. Connect a ribbon cable to the port. Use a continuity tester to locate and identify the following pins: pin 1 (PA0), pin 5 (PA4), pin 6 (PA5), pins 9 and 10 (5 Volts), pins 11 and 12 (Ground). (See page 15 of the User's guide that came with your board.)

4. Make two test light emitting diode probes. It is a good idea to have several probes handy. To make a test probe, connect a 3.3K resistor to the **anode** of a light emitting diode (See figure 3.1). Use a light emitting diode with an operating voltage of 1.7 volts and a current rating in the 1-5 mA range. If you use an light emitting diode with 20+ mA operating current, the light emitting diode would be dim when it lights up and you may have to look carefully to see if it is on. Test the probe by connecting the free end of the resistor to pin 9 and the free end of the light emitting diode to pin 11. The light emitting diode should light-up. If not, the chances are you have connected the resistor to the cathode of the light emitting diode. Redo your circuit by connecting the resistor to the anode.

5. Connect a (LEFT) test probe between pin 5 and pin 11. The cathode of the light emitting diode should be connected to pin 11 and the free end of the resistor to pin 5.

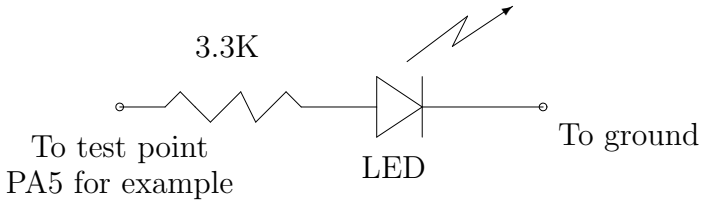

Figure 3.1: *Test Probe. Make sure that the resistor is connected to the anode of the light emitting diode. The longer lead of the light emitting diode is the anode. To test a pin, connect the cathode to ground and the resistor to the pin.*

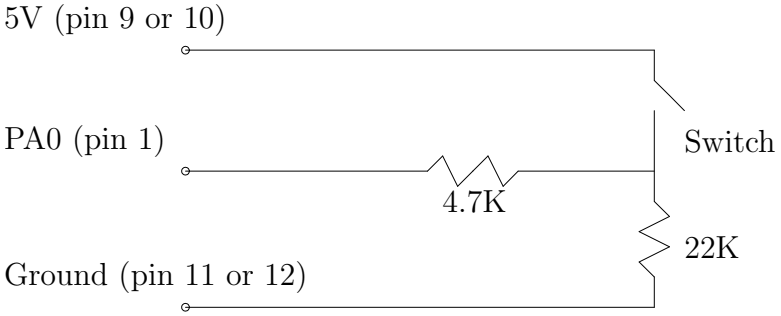

Figure 3.2: *Typical input connection*

6. Connect a (RIGHT) test probe between pin 6 and pin 12. The cathode of the light emitting diode should be connected to pin 12 and the free end of the resistor to pin 6.

7. Label the two light emitting diodes as LEFT and RIGHT so that it is easy to identify them (you can use a magic tape and small pieces of paper).

8. Connect the input circuit as shown in figure 3.2.

9. Check and recheck all your connections before you connect the power to the HC11.

Special instructions for `FOX11` users

1. Connect two LED circuits shown in figure 3.1 to PA4 and PA5. Label the circuit connected to PA4 `LEFT` and the one connected to `PA5` as `RIGHT`.

2. Connect the input circuit as shown in figure 3.2.

3.10.2 Controlling the LED

Use memory modify, `MM $1000`, command to modify `PORTA`. Change the value in the location to `00`, `10`, `20`, and `30`. (Note: When you communicate directly with the HC11 using BUFFALO commands, you do not type the `$`.) After each change, look at the state of the light emitting diodes and write down what you see. Provide a brief explanation of what you see.

Using the methods you learnt in earlier labs, write a program that will read the keyboard and depending on what the user types, perform the following:

Key	Action
`Q` or `q`	Turn on the LEFT led. The state of other led should not change.
`Z` or `Z`	Turn off the LEFT led. The state of other led should not change.
`E` or `e`	Turn on the RIGHT led. The state of other led should not change.
`C` or `c`	Turn off the RIGHT led. The state of other led should not change.

3.10.3 Reading an external switch

Use memory dump to see the contents of `$1000`. Close the input switch and dump the contents of location `$1000`. Repeat the experiment with the switch open. Write down what you see and provide a brief explanation of your observation.

Write a program that will, in an infinite loop, print either `CLOSED` or `OPEN`, depending on the state of the switch. The program should continually monitor the state of the switch, and print `CLOSED` is the switch is closed. Or else it should print `OPEN`.

Chapter 4

Programs

4.1 Objective

To become familiar with writing programs. This lab also illustrates the use of assembler directives.

4.2 What you should do

You will have to turn in your `LST` files for the programs you write for this lab. Check with the TA for additional instructions.

4.3 Programs

When you write programs in a high level language such as C, C++, Java, the compiler facilitates modular program development using function. In an earlier lab, we discussed some basic rules for writing functions. Another facility that higher level languages provides is the use of variables. Key features of variables in high level languages are:

1. Variables generally have a name and you use the name to manipulate them.
2. Variables have a type. The compiler keeps track of the type and makes sure that the way a variable is used conforms to its type.
3. Compiler negotiates with the operating system to obtain adequate memory for the variable.
4. Compiler makes sure that a memory allocated to a variable is not accidentally also allocated to another variable

5. Compiler keeps track of the life and scope of the variable so that the variable is available only when it is in the scope of the instruction (the simplest case is the distinction between global and local variables.

In assembly code, you are pretty much on your own. First of all, variables are known by their address. Some variables require more than one location. In this case, you allocate consecutive memory locations for the variable, and the address of the variable is the first of these locations. It is your responsibility to make sure that (a) you set aside adequate memory for the variable, and (b) you do not assign the same location for more than one variable. The most common errors that programmers make is not keeping this in mind. For example, the programmer may set the address of a two byte variable as `$3120` and the address of another variable as `$3121`. **It is a bad idea for you to manually assign addresses to variables.** Let the assembler do it for you. However, if you have to manually assign an address to a variable, `EQU`ate a label to the variable as

```
TOTAL EQU $3800 *variable to keep track of the total
```

The above instruction defines a variable called `TOTAL` that is stored in locations starting from `$3800`. Note you have no idea how many bytes are needed for the variable and therein lies the potential bug!

4.3.1 How the assembler works

Inside the assembler is a variable called the *location counter*, also known as the `dot` in the unix community. When the assmbler starts, the location counter is initialized to zero. As the assembler reads your program, the location counter changes in response to your code. For example, the `ORG` command sets the location counter to the value after the `ORG` instruction. Thus, if you want to change the location counter to `$00EB`, you would write

```
ORG $00EB
```

The assembler keeps a copy of the HC11 memory internally, and the location counter is used to address the memory. Here, briefly, is how the assembler works:

1. Read the instruction. If the instruction has a label, and it is not an `EQU` instruction, then equate the label to the location counter. If the instruction is an `EQU` instruction, the label is `EQU`ated to value after the mnemonic.

2. If the instruction is a valid HC11 instruction, generate the code for the instruction. The code is stored in consecutive memory locations starting from the address given by the location counter. The location counter is incremented by the amount of memory needed to store code.

3. If the instruction is a `RMB` instruction, then add the value after the mnemonic to the location counter.

4. If the instruction is a `FCB` command, then a sequence of comma-separated values following mnemonic are stored in consecutive locations starting from the address given in the location counter. The location counter is incremented by the amount of memory needed to store these bytes. `FCC` is a convenient way to specify a sequence of ASCII characters. Thus the following two statements are equivalent

   ```
   FCB 72, 69, 76, 76, 79
   FCC /HELLO/
   ```

   The `FDB` instruction is similar to `FCB`, except the values are interpreted as 16-bit numbers. *These initializations are done in your PC and then transferred to the HC11 via the* `S19` *file. If these memory locations in your HC11 are modified after the transfer, either by your program or by accident, then you have to reload your* `S19` *file! Also, DO NOT use these commands to initialize variables. Your program will work only once. If you rerun your program, the variables will not be reinitialized!*

Here is an example of defining variables and constants (variables that your program will not modify).

```
        ORG $3000 ;Start of data section.
V1      RMB 4 ; set aside 4 bytes. EQUate V1 to first address
V2      RMB 11; set aside 11 bytes
THOU    FCB 3, $E8 ; Initialize 16-BIT variable called THOU
BUFF    FILL $22,18 ; same as fcb with $22 repeated 18 times
BUF2    RMB 20
        OPT s ; turn on symbol dump option
```

It is a good idea to turn on the symbol dump option. This will cause assembler to print all the symbols at the end of your program listing. Type in the above sequence of instructions and assemble it. Look at the `LST` file and write down what the symbols `V1`, `V2` etc., are `EQU`ated to and explain the results.

4.4 Your first program

We will now write the first program. When writing assembly code, a convenient way to document your code is to write the *pseudocode*. Rather than invent another

pseudo language, I will use a C like syntax[1]. The program we want to write should be similar to the following C program (since HC11 is essentially an 8-bit micro, all variables will be unsigned chars to keep things simple).

```
#include <stdio.h>
unsigned char v1, v2, v3, v4; unsigned char
total; main() {
    v1 = 11;
    v2 = 0x2F; /* In C prefix 0x denotes HEX */
    v3 = 'A';
    v4 = 044; /* In C prefix 0 denotes octal */
    total = v1+v2+v3+v4;
    printf("%02X", (unsigned) total);
    return 0;
}
```

Compile and run the above program using any C compiler. The output of your program should be `9F`. Now assemble the following HC11 program, and run it to verify that you get the same answer:

```
;Name: etc
;
; This program adds 4 numbers and prints the answer.
;
outlhlf     equ $ffb2
outrhlf     equ $ffb5

    org $2000
main
    ; v1 = 11;
    ldaa #11
    staa v1

    ;v2 = 0x2f;
    ldaa #$2f
    staa v2

    ;v3 = 'A';
    ldaa #'A'
    staa v3
```

[1]This reverses history! C language was invented to avoid writing assembly code!

```
    ;v4 = 044;
    ldaa #@44
    staa v4


    ;total = v1+v2+v3+v4;
    ldaa v1
    adda v2
    adda v3
    adda v4
    staa total

    ;print total as 2 digit hex number
    ldaa total
    jsr outlhlf
    ldaa total
    jsr outrhlf

    swi

    org $3000
v1 rmb 1
v2 rmb 1
v3 rmb 1
v4 rmb 1
total rmb 1

   opt s
```

Modify the C program so that `v3 = 'A';` is replaced by `v3=getchar();`. Run the C program and type the upper case letter `A` and press enter. You should see the same output as before. Now replace the instruction `ldaa #'A'` by `jsr inchar` and make sure that you equate `inchar` to `$ffcd`. Run the assembly above assembly language program and verify that it behaves the same as the C program.

4.5 On your own!

1. Write an assembly program that does the same as the following C program:

```
#include <stdio.h>
```

```
char v;
char total;
main() {
    v  = 0;
    total = 0;

    v = v+1;
    total = total + v;


    v = v+1;
    total = total + v;

    v = v+1;
    total = total + v;

    v = v+1;
    total = total + v;

    v = v+1;
    total = total + v;

    printf("%02X", (unsigned) total);
    return 0;
    }
```

2. Modify the program so that the initialization `v = 0` is replaced by a call to `getchar()`. Run your C program and try different inputs at the keyboard. Make corresponding changes to the assembly language program and verify your results.

3. Modify the assembly language program so that you use a counting loop to loop 5 times over the basic code.

4. Modify the previous version (using a loop) where the number of times around the loop is in an 8-bit *variable* called `count`. Your program should initialize the variable to `5` so that the loop is executed 5 times.

5. A convenient way to get a value between 0 and 9 from the user is to use the instruction sequence

```
jsr inchar
```

```
    anda #$0F
    ; in C, use: getchar() & 0X0F
```

Use the above sequence to let the user specify the count. When expecting the user to enter a value, it is a good idea to prompt the user. Your code may look something like:...

```
;; various equates and comments...
;
;
    org $2000
    ldx #prompt
    jsr outstrg
    jsr inchar
    anda #$ff
    staa count
    ;
    ; rest of the code goes here


    org $3000
    ; various RMB
    count RMB 1
    v   RMB 1
    total RMB 1

prompt fcc /How many times please? /
    fcb 4
```

6. Write a program that will do the following:

   (a) Get a number between 0 and 9 from the user and store in a variable called `v1`.

   (b) Multiply `v1` by `4` and store the result in a variable called `v2`.

   (c) Add `v1` and `v2` and store the result in a variable called `v3`.

   (d) Multiply `v3` by `2` and store the result in a variable called `v4`.

   (e) Print the values in the variables separated by a comma. In the last lab, you saw an example for printing a colon. You can use the same approach.

(f) Modify the program so that the output reads some thing like:

```
v1 = 07
v2 = 1C
v3 = 23
v4 = 46
```

7. Write a function that will be passed a value in **A** register. The function should return 10 times the value passed to it. Use the above function to write a program that does the following: The program should get a digit from the user, multiply it by 10 and store the value in a variable called `tens`. It should then get a digit from the user and add it to the variable `tens` and store it in a variable called `DecimalIn`. The program should then print the value in the variable.

8. Use the code you wrote to create a function that will read a two digit decimal number from the user. Call this function `ReadDecimal`. Write a program that will call this function to read a two digit number and store the value in a variable called `DecimalIn`. The program should then print the value in the variable.

9. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

```
#include <stdio.h>

char delta, value;
char count;
main() {
    value = 0;
    delta = 1;
    count = 11;

foo:
    if (count == 0) goto bar;

    printf("%02X", (unsigned) value);
    putchar(','); putchar(' ');

    value = value + delta;
    delta = delta + 2;
    count = count -1;
```

```
    goto foo;

bar:
    return 0;

}
```

10. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

```
/*

   I decided to use fprintf(stderr instead of printf(
   as printf( and getche don't mix well.

   I am forced to use getche instead of getchar because
   getchar    puts the terminal in the 'cooked' mode rather
   than 'raw' mode    and there seems to be no way to uncook
   the input in MS Windows.

   The program as written will work on all variants of Windows
   and on unix boxes if you know the right curses!

   Note: getche is an exact equivalent of INCHAR in BUFFALO
         To print a string, you use OUTSTRG in BUFFALO

*/

#include <stdio.h>
#include <conio.h>

char c;

main() {
foo:
    fprintf(stderr,"\n\nWelcome! Your choices:\n\n");
    fprintf(stderr,"\n1. Set temperature");
    fprintf(stderr,"\n2. Set Speed\n"
    fprintf(stderr,"\nChoice please: ");
    c = getche();
```

```
    if (c == '1') goto one;
    if (c == '2') goto two;
    fprintf(stderr,"\n\nNot a valid choice!\n");
    goto foo;

one:
    fprintf(stderr,"Good choice. \n");
    goto more;
two:
        fprintf(stderr,"Try later\n");


more:
    fprintf(stderr,"Try again? ");
    c = getche();
    if (c == 'y') goto foo;
    if (c == 'Y') goto foo;
    if (c == 'n') goto bye;
    if (c == 'N') goto bye;
    goto more;

bye:
    return;
}
```

Chapter 5

Tables

5.1 Objective

To become familiar with table driven code.

5.2 What you should do

You will have to turn in your `LST` files for the programs you write for this lab. Check with the TA for additional instructions. In most of the examples, the name is left blank as `_____________`. Make sure you enter your name in its place.

5.3 Tables

What is a table? Technically, table is same as an array except we use the term *table* to refer to arrays whose elements are constants. To operate on the table, we need two pieces of information, *where, in memory, the table starts(* **the starting address***)* , and *how many elements are in the table (* **the size or dimension***)*. In this lab, we will initially consider the case where each element requires one byte of memory.

5.4 Setting up a table

To setup a table in memory, we generally use `FCB`, `FDB` and `FCC`. For example, if we want to setup a table of prime numbers, we would write (using decimal since it is easier to read):

...

```
    ORG $3000 *data section
...
...
primes fcb 2,3,5, 7, 11, 13, 17, 19, 23, 29 *table of some primes
nprimes equ 10 *number of entries in the table
```

If you want to set up a table of ascii code for digits, you would write

```
...
    ORG $3000 *data section
...
...
digits1 fcb  $30, $31, $32, $33, $34, $35, $36, $37, $38, $39
ndigits1 equ 10 *number of entries in the table
```

A **better** way to do the same is to write

```
...
    ORG $3000 *data section ..$
...
...
digits fcc  /0123456789/
ndigits equ 10 *number of entries in the table
```

Assembler will convert `FCC` to a sequence of `FCB`.

Exercise: Write an ASM file with the two versions of the digits tables, assemble the file and look at the `LST` file. Verify that the two tables are identical.

5.5 Working with tables

We will write functions to perform some basic tasks with the tables. In all these functions, we will use the following convention:

1. The starting address will be passed to the function in the **X** register.
2. The size of the table will be passed to the function in the **B** register.

5.5.1 Table lookup

This is the simplest and the most useful function. We want to know if an element is in the table. For now, we will work with a table of 8-bit quantities. The value to be looked up will be passed in the **A** register. The function will have to return

a `Yes/No` value. A convenient way to return a `Yes/No` value is to use a hardware flag. Let us use the **C**arry flag. The function will set the flag if the answer is yes; or else it will clear the carry.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; lookup: Function to lookup a value in a table
;         Checks if the value in A register is in the table
;
; Entry: Starting address in X, size in B, value in A
; Exit:  Carry set if the value in A is in the table;
;              cleared if not in the table
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

lookup
lkploop
        tstb
        beq     notthere *this is the basic counting loop
        cmpa    0,x
        beq     foundit
        inx
        decb
        bra     lkploop
notthere
       clc
       rts
foundit
       sec
       rts
```

Here is a program that uses the function:

```
;Name:
;email:
;date:
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

ucase       equ $ffa0
wchek       equ $ffa3
dchek       equ $ffa6
```

```
init        equ $ffa9
input       equ $ffac
output      equ $ffaf
outlhlf     equ $ffb2
outrhlf     equ $ffb5
outa        equ $ffb8
out1byt     equ $ffbb
out1bsp     equ $ffbe
out2bsp     equ $ffc1
outcrlf     equ $ffc4
outstrg     equ $ffc7
outstrgo    equ $ffca
inchar      equ $ffcd
vecinit     equ $ffd0

    org $3000
; setup some strings ...
preamble
    fcc /====================================/
    fcb 10
    fcc /Lab on using Tables/
    fcb 10, 10 *use 10 to start a new line
    fcc /Name: _____________________/
    fcb 10
    fcc /This program is an infinite loop! /
    fcb 10
    fcc /Hit the reset button to quit/
    fcb 10
    fcc /====================================/
    fcb 10,10,10, 4

yesstr  fcc /  is a vowel/
    fcb 10, 4
nostr   fcc /  is not a vowel/
    fcb 10, 4

; setup the table of vowels

vowels  fcc /aeiouAEIOU/
nvowels equ 10
```

```
    org $2100
    ldx #preamble
    jsr outstrg
mainloop
    jsr inchar
    ldx #vowels
    ldab #nvowels
    jsr lookup
    bcs isvowel
    ldx #nostr
    jsr outstrgo
    bra mainloop
isvowel ldx #yesstr
    jsr outstrgo
    bra mainloop


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; lookup: Function to lookup a value in a table
;         Checks if the value in A register is in the table
;
; Entry: Starting address in X, size in B, value in A
; Exit:  Carry set if the value in A is in the table;
;              cleared if not in the table
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

lookup
lkploop
        tstb
        beq     notthere *this is the basic counting loop
        cmpa    0,x
        beq     foundit
        inx
        decb
        bra     lkploop
notthere
       clc
       rts
foundit
       sec
       rts
```

Exercise: Type the above program, assemble it, transfer the `S19` file to the 68HC11, and run the program with `CALL 2100`. When the program starts, type the following text `Pack my box with five dozen liquor jugs`.

Exercise: The lookup program affects the **X** and **B** registers. Modify the function so that the function initially stores these two registers in the stack, and restores them before returning. Verify that the program works correctly.

Exercise: Modify the program by adding another table, the table of symmetric characters: `AHIMOTUVWXYimnouvwxy`. The program should, in addition to checking to see if a character is a vowel, it should also check to see if it is a symmetric character. Run the program and enter the following text: `Axiomboard`. You should get an output similar to the following:

```
 done
 >c 2100

 ====================================
Lab on using Tables

Name: ____________________
This program is an infinite loop!
Hit the reset button to quit
====================================


A  is a vowel and is a symmetric character
x  is not a vowel and is a symmetric character
i  is a vowel and is a symmetric character
o  is a vowel and is a symmetric character
m  is not a vowel and is a symmetric character
b  is not a vowel and is not a  symmetric character
o  is a vowel and is a symmetric character
a  is a vowel and is not a  symmetric character
r  is not a vowel and is not a  symmetric character
d  is not a vowel and is not a  symmetric character
```

5.5.2 Input with validation

Another use of the table lookup is to validate input from the user. Suppose we want the user to enter a social security number. In this case, we want to make sure

that we accept only digits and ignore non-digits (the user may be in the habit of typing spaces, dashes etc. You want to silently ignore these). So it will be useful to write a function, called `rddigit` that will accept only digits. The following program shows a typical usage:

```
;Name:
;email:
;date:
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

ucase       equ $ffa0
wchek       equ $ffa3
dchek       equ $ffa6
init        equ $ffa9
input       equ $ffac
output      equ $ffaf
outlhlf     equ $ffb2
outrhlf     equ $ffb5
outa        equ $ffb8
out1byt     equ $ffbb
out1bsp     equ $ffbe
out2bsp     equ $ffc1
outcrlf     equ $ffc4
outstrg     equ $ffc7
outstrgo    equ $ffca
inchar      equ $ffcd
vecinit     equ $ffd0

    org $3000
; setup some strings ...
preamble
    fcc /====================================/
    fcb 10
    fcc /Lab on using Tables/
    fcb 10, 10 *use 10 to start a new line
    fcc /Name: _____________________/
    fcb 10
    fcc /This program is an infinite loop! /
    fcb 10
```

```
    fcc /Hit the reset button to quit/
    fcb 10
    fcc /====================================/
    fcb 10,10,10, 4


; setup the table of digits

digits  fcc /0123456789/
ndigits equ 10


    org $2100
    ldx #preamble
    jsr outstrg
mainloop
    jsr rddigit
    bra mainloop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; rddigit: Behaves like inchar, except ignores non-digits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rddigit jsr input *This does not wait for the user
    tsta
    beq rddigit *Looks like the user has not typed anything
;
; if we get here, the user typed something. Verify it first
;
    ldx #digits
    ldab #ndigits
    jsr lookup
    bcc rddigit *oops, not in the table. Go back for more
;
; if we get here, the input was ok
; echo it back as the user would like some feedback
;

    jsr     outa
    rts
```

```
;;; Add the code for lookup function here
```

Exercise: Type the above program and run it. Enter the following input and explain what you see:

Exercise: Type the above program and run it. Enter the following input and explain what you see: `123-34-1879`.

Exercise: Modify the above program so that it accepts exactly 10 digits and stops with an `SWI` after reading 10 characters. In other words, convert the main loop into a counting loop. *You should keep the count in the* **B** *register.* Verify your program with the input: `823--xx-34-1879`.

5.5.3 Translations using tables

Some time we want to translate a value to another value. For example if the user types the character `8`, your program will receive the ascii code for the character. You now will have to convert it to its value, viz `8`. This is easy since you just have to subtract `$30`. However, if the user enters data in HEX, then he would expect the program to translate `A` and `a` to 10, `B` and `b` to 11 and so on. For problems of this nature, we first write down the translation table

value	translation
$30 or '0'	0
$31 or '1'	1
$32 or '2'	2
etc.	etc
$39 or '9'	9
$41 or 'A'	10
$61 or 'a'	10
$42 or 'B'	11
$62 or 'b'	11
etc.	etc
etc.	etc

In our assembly program we set up two tables. It is extremely important that the two tables be ordered as follows: The table of values first immediately followed by the table of translations. For example, to perform the above translation, we will write

```
hexchars fcc /0123456789AaBbCcDdEeFf/
hextrans fcb 0,1,2,3,4
         fcb 5,6,7,8,9
         fcb 10,10, 11,11, 12,12, 13,13, 14,14, 15,15
nhexchars equ 22
```

To illustrate the use of the translation table, let us write a function that will translate telephone numbers. For example, if the input to the program is a mixture of numbers and letters as in `1-800-CALLATT` the program should translate it to `1-800-2255288`. The translation table can be found on any telephone and is:

value	translation
'A', 'B', 'C'	2
'D', 'E', 'F'	3
'G', 'H', 'I'	4
'J', 'K', 'L'	5
'M', 'N', 'O'	6
'P', 'Q', 'R', 'S'	7
'T', 'U', 'V'	8
'W', 'X', 'Y', 'Z'	9

The following program sets up the above table and uses it to translate phone numbers.

```
;Name:
;email:
;date:
;
; Standard buffalo equates
; Make sure you have ALL the equates in the file.
;

ucase       equ $ffa0
wchek       equ $ffa3
dchek       equ $ffa6
init        equ $ffa9
input       equ $ffac
output      equ $ffaf
outlhlf     equ $ffb2
outrhlf     equ $ffb5
outa        equ $ffb8
out1byt     equ $ffbb
out1bsp     equ $ffbe
```

```
out2bsp     equ $ffc1
outcrlf     equ $ffc4
outstrg     equ $ffc7
outstrgo    equ $ffca
inchar      equ $ffcd
vecinit     equ $ffd0

    org $3000
; setup some strings ...
preamble
    fcc /====================================/
    fcb 10
    fcc /Lab on using Tables/
    fcb 10, 10 *use 10 to start a new line
    fcc /Name: _____________________/
    fcb 10
    fcc /This program is an infinite loop! /
    fcb 10
    fcc /Hit the reset button to quit/
    fcb 10
    fcc /====================================/
    fcb 10,10,10, 4


; setup the tables
alphabet fcc /ABCDEFGHIJKLMNOPQRSTUVWXYZ/
nums     fcc /222333444555666777788899999/
nalphabet equ 26


    org $2100
    ldx #preamble
    jsr outstrg
    jsr outcrlf

mainloop
    jsr inchar
    jsr ucase ; convert to upper case if necessary
    jsr translate ; perform the translation
    psha          ; save it in the stack for now
```

```
; print 2 spaces so the output looks neat
    ldaa    #' '
    jsr     outa
    jsr outa

    pula        ; get it back
    jsr outa
    jsr outcrlf
    bra mainloop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;translate: Translates telephone codes.
;Entry: Register A has the value to be translated
;Exit: If A has an upper case letter, then its content is replaced
;      by the the translation given on the phone is performed
;          ABC -> 2, DEF ->3, etc.
;      If A has any other character, it is left alone.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

translate
    ldx #alphabet
    ldab #nalphabet
    jsr lookup
    bcc bye ; Not an alphabet. Leave it alone

    ldaa nalphabet,x ; This is the key to translation
bye
    rts



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; lookup: Function to lookup a value in a table
;         Checks if the value in A register is in the table
;
; Entry: Starting address in X, size in B, value in A
; Exit:  Carry set if the value in A is in the table;
;              cleared if not in the table
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

lookup
```

```
lkploop
        tstb
        beq     notthere *this is the basic counting loop
        cmpa    0,x
        beq     foundit
        inx
        decb
        bra     lkploop
notthere
       clc
       rts
foundit
       sec
       rts
```

Exercise: Type the above program and enter the input: `1-800-UMD-ALUM`. You should see the following output. Clearly explain how the translation gets done.

```
 ====================================
Lab on using Tables

Name: ____________________
This program is an infinite loop!
Hit the reset button to quit
====================================



 1  1
 -  -
 8  8
 0  0
 0  0
 -  -
 U  8
 M  6
 D  3
 -  -
 A  2
```

```
L  5
U  8
M  6
```

Chapter 6

Timing using Polling

6.1 Objective

Introduces polling timer overflow flag to create a simple clock and to generate `ON-OFF` output signals.

6.2 Getting started

Make sure you have made the external connections show in figures 3.1 and 3.2 before proceeding further.

6.3 Timing

In this part of the lab, we will monitor the `TOF` flag. This flag is controlled by the *free running counter*. The free running counter is a 16 bit counter that counts the clock ticks. It is set to `$0000` on power up. It counts up to `$FFFF`and then rolls over to `$0000` and the process is repeated until you power off the HC11. Every time the counter rolls over, it sets the `TOF` flag. This flag is not automatically reset and it is your responsibility to reset it in your code depending on your need. To reset the flag you have to write a 1 to it [1].

Your HC11 most likely uses a 8 MHz crystal which means that the processor speed is 2 MHz, or you get 2×10^6 clock ticks every second. The free running counter rolls over every 2^{16} ticks, or you get

$$\frac{2 \times 10^6}{2^{16}} = 30.52 \text{ overflows every second}$$

[1] This is true of all HC11 flags. You reset a flag, i.e. make it go to zero, by writing a 1 to it!

We can use this to create a $\frac{30.52}{2}$ Hz square wave by toggling an output pin every time counter rolls over. Here is the code (you have to fill in the details!)

```
; Various defines go here ...
       ORG $D000 don't forget the $
ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $C000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG

LOOP1
; CLEAR THE FLAG. NO HARM IS DONE IF IT IS ALREADY CLEARED
       LDAA #%10000000
       STAA TFLAG2

;WAIT FOR THE FLAG TO BE SET
LOOP2
      LDAA TFLAG2
      ANDA #%10000000
      BEQ LOOP2

;NOW TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

;DO IT ALL OVER AGAIN
      BRA LOOP1
```

Type and run the above program. Connect the LED to PA4 (pin #5) and you should see it flicker. Connect the pin to a oscilloscope and verify that you are generating a square wave. Verify that the frequency is correct.

6.3.1 Slowing it down

In the last experiment, we toggle the pin so fast that chances are you did not notice the LED's blink. We can slow it down by toggling only every so many TOF overflows (say 31 overflows). All we need to do is introduce a counting loop as shown below. Note that the program structure is not changed at all.

```
; Various defines go here ...
       ORG $D000 don't forget the $
ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $C000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG
LOOP1


      LDAB #31 'SET UP COUNTER
LTOP
      TSTB
      BEQ LBOT


; CLEAR THE FLAG. NO HARM IS DONE IF IT IS ALREADY CLEARED
       LDAA #%10000000
       STAA TFLAG2

;WAIT FOR THE FLAG TO BE SET
LOOP2
      LDAA TFLAG2
      ANDA #%10000000
      BEQ LOOP2

      DECB
      BRA LTOP

LBOT
```

```
;NOW TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

;DO IT ALL OVER AGAIN
      BRA LOOP1
```

Chapter 7

Interrupt Processing

7.1 Objective

To become familiar with interrupt processing.

7.2 Background

In an earlier lab, you had to generate a 30.52 Hz square wave signal on PA4 pin. The code for generating the square wave is given below for your reference. Make sure that you run the program and verify that you get the square wave before proceeding further. Also, to understand this lab, you must connect the PA4 pin to a oscilloscope and see the square waves.

```
 ; Various defines go here ...
       ORG $3000 don't forget the $

ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $2000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG

LOOP1
```

```
; CLEAR THE FLAG. NO HARM IS DONE IF IT IS ALREADY CLEARED
       LDAA #%10000000
       STAA TFLAG2

;WAIT FOR THE FLAG TO BE SET

LOOP2
      LDAA TFLAG2
      ANDA #%10000000
      BEQ LOOP2

;NOW TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

;DO IT ALL OVER AGAIN
      BRA LOOP1
```

7.3 Interrupts

If you study the above code, most of the time is spent waiting for the clock to rollover (the loop at `LOOP2`). This is a lot like sitting in front of the clock and watching and waiting for the clock to rollover. Or, for that matter, sitting in front of a stove and watching and waiting for kettle to boil, or watching food being cooked in a microwave oven. A better solution would be to go about ones job and arrange matters so that one is told when an event occurs (the clock chimes, the kettle whistles, the microwave oven sounds an alarm etc.). When we are told that the event has occurred, we then take appropriate action (turn off the stove, take food out of the oven etc.). Most of the HC11 interrupts work the same way. In essence this is what happens:

1. When an event occurs, a flag is set. For example, the clock rollover sets the `TOF` flag.

2. Associated with the flag is a masking bit. The name of the bit is the same as the flag except the final `F` is replaced by `I`. The mask associated with `TOF` is `TOI`.

   (a) If the mask is zero, then nothing much happens. The event is ignored by the interrupt processing structure.

(b) However, if the mask is set, then request for service is generated.

i. The `I` bit in the CCR is a master disable switch. If this is set (by using the command `SEI`), then the request for service does not interrupt the computer and is hence ignored.

ii. However if `I` bit is cleared (by using the command `CLI`), then the CPU is interrupted.

Note: It is extremely important that you have an `SEI` before any code that can turn on an interrupt and and `CLI` after all relevant and required initialization is performed.

3. When a CPU is interrupted, it stops its current task and starts the service.

4. When performing the service, you get a completely new set of registers. So you can not assume that the registers will have any specific value. Also, when the service terminates, the new set of registers is destroyed. So you can not assume that the rest of the code can see what you stored in the register as part of the service. In fact, the interrupted task would be oblivious to the fact that a service was provided. The only way it can find out is if the service modifies some memory location.

5. The location of the service that is associated with a particular interrupt is defined by the hardware manufacturer, and is called the *jump vector*. This would be in read only memory and can not be chanted. The operating system, BUFFALO, sets the start of the service to a known location and sets aside three bytes at that location. Your service will start with `JMP` instruction to the actual code for the service. Your service **must end with the `RTI` instruction**.

   The address of services to three important interrupts is given in the following table:

Interrupt	**Service location**
TOF	$00D0
RTIF	$00EB
OC2F	$00DC

6. Your service code should, as part of the service, *turn off* the flag that generated the interrupt. If not, the request for service will still be active and will generate a new service request as soon as the current service end!

Here is a short checklist for what you should do:

1. In your main routine, enable an interrupt by turning on the associated mask.

2. Write the service routine. As part of the service make sure you turn off the flag that generated the interrupt.

3. Let HC11 know where to find the service. In other words, Link the service to the request.

With this background, we will modify the square wave generator to use the interrupt. Here is the complete code with some of the standard equates left out (you need to have them at the top of the file!). You should compare this code with the earlier one. In this code, the main program does nothing really interesting. It just prints a series of Z's to the screen

```
 ; Various defines go here ...
       ORG $3000 don't forget the $

ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $2000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG

LOOP1

; Enable TOF interrupt by setting TOI (bit#7 in TMSK2)
       SEI
       LDAA #%10000000
       STAA TMSK2
       CLI

; Now go about your business of printing Z's
       LDAA #'Z'
LOOP   JSR OUTA
       BRA LOOP

; End of main program

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; INTERRUPT SERVICE

SERVICE

; TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

; TURN OFF THE FLAG!
      LDAA #%10000000
      STAA TFLG2

; END WITH AN RTI
      RTI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Connect the service to the interrupt ;

     ORG $00D0 ; $00D0 WHERE THE SERVICE STARTS
     JMP SERVICE ; JUMP TO WHERE THE SERVICE CODE ACTUALLY IS
```

How does this code differ from the previous one? We don't wait for the clock to rollover. Instead, the main program goes about its task (in this case not a very interesting one!). Note that although the service routine uses the **A** register, this does not affect the value the main routine has stored in the **A** register (the character `Z`).

7.4 The Real time interrupt

The real time interrupt, `RTI` (not to be confused with the `RTI` instruction) acts exactly like the timer overflow interrupt, except you can control the time between interrupts using the last two bits (0 and 1) of `PACTL` at location `$1026`. If the both the bits are set, the time between the interrupts is is 32.768 ms, i.e. same as timer overflow. However, you can decrease the time between the interrupts (increase the rate) by changing the last two bits as `PACTL` as shown below:

Last two bits of PACTL	Time between interrupts
00	4.096 ms (244.1 Hz)
01	8.192 ms (122 Hz)
10	16.384 ms (61 Hz)
11	32.768 ms (30.5 Hz)

Thus if we want to use the RTI interrupt, we have to change the above code as shown below. Note the crucial differences:

```
 ; Various defines go here ...
       ORG $3000 don't forget the $

ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $2000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG

LOOP1

; Enable RTIF interrupt by setting RTII (bit#6 in TMSK2)
       SEI
       LDAA #%01000000 <= This is different
       STAA TMSK2

       LDAA #%00000011 <= THIS IS NEW
       STAA PACTL

       CLI

; Now go about your business of printing Z's
       LDAA #'Z'
LOOP   JSR OUTA
       BRA LOOP

; End of main program
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERRUPT SERVICE

SERVICE

; TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

; TURN OFF THE FLAG!
      LDAA #%01000000 <= This is different
      STAA TFLG2

; END WITH AN RTI
      RTI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Connect the service to the interrupt ;

     ORG $00EB ; $00EB0 WHERE THE SERVICE STARTS
     JMP SERVICE ; JUMP TO WHERE THE SERVICE CODE ACTUALLY IS
```

7.4.1 Exercises

1. Modify the above program to generate a 61 Hz square wave by setting the RTI interrupt rate to 61 Hz.

2. After you made the modification, toggle PA4 every 61 interrupts (Hint, set up a counter and initialize it to 61 in the main program. In the interrupt service, decrement the counter. When the counter reaches zero, toggle the pin and reset the counter back to 61). Verify that the signal you generate is a 1 Hz square wave

3. Create a simple clock. In addition to toggling the pin, increment an 8-bit variable called `TIME`. In the main loop, instead of printing `Z`'s, print the variable using `OUT1BSP`.

7.5 The output compare interrupt

The HC11 has 5 `OCx` interrupts. These are like alarm clocks. You set a desired 'alarm' time and when the clock matches the alarm setting, the `OCxF` flag will be turned on and could then generate a request for service. Note that if you do not change the alarm setting, you will still get an interrupt every 32.768 ms. However, having the alarm gives you greater flexibility. First a code that does not change the alarm setting and hence generates 30.5 Hz square wave.

```
 ; Various defines go here ...
       ORG $3000 don't forget the $

ME     FCC /Your name/
       FCB 10
       FCC /ECE 372/
       FCB 10
       FCC /Date the program was last changed/
       FCB 10, 10, 4

       ORG $2000 DONT FORGET THE $
       LDX #ME
       JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG

LOOP1

; Enable OC2 interrupt by setting OC2I (bit#6 in TMSK1)
       SEI
       LDAA #%01000000 <= This is different
       STAA TMSK1

      CLI

; Now go about your business of printing Z's
       LDAA #'Z'
LOOP   JSR OUTA
       BRA LOOP

; End of main program

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERRUPT SERVICE
```

```
SERVICE

; TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

; TURN OFF THE FLAG!
      LDAA #%01000000 <= This is different
      STAA TFLG1

; END WITH AN RTI
      RTI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Connect the service to the interrupt ;

     ORG $00DC ; $00DC WHERE THE SERVICE STARTS
     JMP SERVICE ; JUMP TO WHERE THE SERVICE CODE ACTUALLY IS
```

Verify that the above code also generates a 30.5 Hz square wave. Now we can reset the alarm to get a different frequency. For example, if we modify the service routine as follows, we will get an interrupt every 2000 clock ticks or every 1 ms for a 1 K Hz signal.

```
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERRUPT SERVICE

SERVICE

; TOGGLE PA4
      LDAA #%00010000
      EORA PORTA
      STAA PORTA

; TURN OFF THE FLAG!
      LDAA #%01000000
      STAA TFLG1
```

```
;set the next interrupt to occur 2000 clock ticks after this

      LDD TOC2 <= This is different
      ADDD #2000
      STD TOC2 <=


; END WITH AN RTI
      RTI
```

Chapter 8

Signal Generation

8.1 Objective

To become familiar with generating square waves.

8.2 Background

In an earlier lab, you had to generate a square wave signal using various interrupts. This lab builds on this. For a general square wave signal, we define the on-time, T_{on}, the off-time, T_{off}, and the period T as shown in the figure 7.1. The goal of the lab is to generate such square waves. The ratio $\frac{T_{\text{on}}}{T}$ is called the `duty cycle` and is expressed as a percentage. When dealing with time it is convenient to talk in terms of clock ticks. The HC11 has a 2 MHz e-clock, or you have 2×10^6 ticks per second. Hence 1 clock tick is 0.5 microseconds.

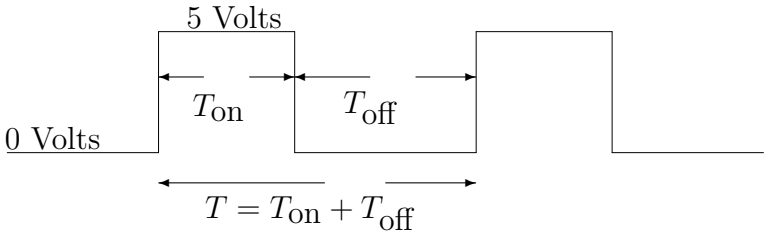

Figure 8.1: 0-5 volt square wave

8.3 Variable frequency signal generator

In this experiment, we will generate a square wave with frequency selected by the user. We will fix the duty cycle at 25%. To get started, we will first write the code for generating a single tone.

8.4 500 Hz tone generator

Here is a typical calculation to generate a 500 Hz signal:

$$\begin{aligned} T &= 1/500 = 2 \times 10^{-3} \text{ seconds} = (2 \times 10^{-3}) \times (2 \times 10^{6}) = 4000 \text{ ticks} \\ T_{\text{on}} &= 4000/4 = 1000 \text{ ticks} \\ T_{\text{off}} &= 4000 - 1000 = 3000 \text{ ticks} \end{aligned}$$

To generate a 50 Hz, 25% duty cycle signal we use two variables called `ONTIME` and `OFFTIME`. Every time we get an `OC2` interrupt, we toggle the pin `PA4` as before. We check to see if we turned the pin `ON` or `OFF`. If we turned it on, then we schedule the next interrupt to occur `ONTIME` ticks later. However, If we turned it off, then we schedule the next interrupt to occur `OFFTIME` ticks later. Thus, the earlier code that was used to generate a square wave is modified as follows:

```
 ; Various defines go here ...
        ORG $3000 don't forget the $

ME      FCC /Your name/
        FCB 10
        FCC /ECE 372/
        FCB 10
        FCC /Date the program was last changed/
        FCB 10, 10, 4

ONTIME  RMB 2
OFFTIME RMB 2

        ORG $2000 DONT FORGET THE $
        LDX #ME
        JSR OUTSTRG ; MAKE SURE YOU HAVE EQU FOR OUTSTRG


; Enable OC2 interrupt by setting OC2I (bit#6 in TMSK1)
; ALSO PERFORM ALL INITIALIZATION BETWEEN SEI/CLI
```

```
        SEI
        LDD #1000
        STD ONTIME
        LDD #3000
        STD OFFTIME

        LDAA #%01000000
        STAA TMSK1

        CLI

; Now go about your business of printing Z's
        LDAA #'Z'
LOOP    JSR OUTA
        BRA LOOP

; End of main program

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERRUPT SERVICE

SERVICE

;       TOGGLE PA4
        LDAA #%00010000
         EORA PORTA
        STAA PORTA

;       TEST TO SEE IF THE PIN WENT HIGH
        ANDA #%00010000
        BNE WENTHIGH

;       NO, PIN WENT LOW. LOAD D WITH OFFTIME
        LDD OFFTIME
        BRA REST

WENTHIGH
;       PIN WENT HIGH. LOAD D WITH ONTIME
        LDD ONTIME
```

```
REST
        ADDD TOC2
        STD TOC2 ; BUMP ALARM SETTING

;       TURN OFF THE FLAG!
        LDAA #%01000000
        STAA TFLG1

        RTI

;       Connect the service to the interrupt ;
        ORG $00DC ; $00DC WHERE THE SERVICE STARTS
        JMP SERVICE ; JUMP TO WHERE THE SERVICE CODE ACTUALLY IS
```

Assemble and run the above program. Connect PA4 to an oscilloscope and verify that the duty cycle and the frequency are correct.

8.5 Variable frequency generator

We now modify the above code to create a variable frequency generator. The program will monitor the keyboard and depending on the number the user enters, it will change the frequency as shown in the table below:

Number	Frequency	Period seconds	Period ticks	On time ticks	Off time ticks
0	440	0.002273	4545	1136	3409
1	466	0.002145	4290	1073	3217
2	494	0.002025	4050	1013	3037
3	523	0.001911	3822	956	2866
4	554	0.001804	3608	902	2706
5	587	0.001703	3405	851	2554
6	622	0.001607	3214	804	2410
7	659	0.001517	3034	759	2275
8	698	0.001432	2863	716	2147
9	740	0.001351	2703	676	2027

As a programmer we are only interested in the on-time and off-time. We use FDB to create two tables in the data section as shown below:

```
ONTIMETBL
    FDB 1136 ,1073, 1013, 956, 902
```

```
    FDB 851, 804, 759, 716, 676
OFFTIMETBL
    FDB 3409, 3217, 3037, 2866, 2706
    FDB 2554, 2410, 2275, 2147, 2027
```

Note that it makes sense to enter numbers in decimal notation. Each entry in the table requires two bytes (we use 16 bit numbers to measure time since the HC11 clock is a 16 bit quantity). Hence we use `FDB` instead of `FCB`. Note that this also means that we have to index through memory in steps of **two bytes**. Suppose we want to access the element #4 in ONTIMETBL and the value #4 is in the **B** register (the number #4 is used only as an illustration. In the application the register **B** will have the value). Then we have to access the element we have to write

```
LDX #ONTIMETBL
ABX
ABX ; NEED TWO ABX'S
??? 0,X
```

We can now modify the single tone generator to a programmable square wave generator by making the following changes:

Before		After
LDD #1000		LDX #ONTIMETBL
STD ONTIME		LDY #OFFTIMETBL
LDD #3000		LDD 0,X
STD OFFTIME		STD ONTIME
		LDD 0,Y
		STD OFFTIME

Before		After
	LDAA #'Z'	LOOP
LOOP	JSR OUTA	JSR INPUT
	BRA LOOP	TSTA
		BEQ LOOP
		ANDA #$0F
		TAB ; B HAS INDEX
		LDX #ONTIMETBL
		ABX
		ABX
		LDY #OFFTIMETBL
		ABY
		ABY
		LDD 0,X
		STD ONTIME
		LDD 0,Y
		STD OFFTIME
		BRA LOOP

8.6 Exercises

1. Make the changes shown above and run the program. Connect `PA4` to an oscilloscope. Press any of the keys `0` to `9` and verify that the frequency changes.

2. Change the program so that when you press any of the keys `0` to `9`, the frequency should be fixed at 100 Hz but the duty cycle changes. Pick 10 different duty cycles. Also, connect the output pin to a digital voltmeter. How does the voltage change with the duty cycle?

Chapter 9

Analog to Digital Conversion

9.1 Objective

To become familiar with Analog to Digital Conversion.

9.2 Background

In this lab you will learn how to use the built in analog to digital converter. In an earlier lab you learnt how to read a digital input on one of the port A pins. The input was either a `1` or a `0` depending on whether the pin voltage was above or below 2.5 volts. In this lab you will learn how to measure a voltage more precisely. HC11 has a built in analog to digital convertor that will measure an analog voltage between a lower limit and a higher limit [1] and convert it to a an 8-bit value. The converted value would be zero if the voltage that is measured is equal to the lower limit and the value will be 255 (biggest 8 bit number) when the voltage is at the upper limit. For voltages in between, the converted value would be linearly related, with the value rounded down to the nearest integer. Thus the converted value is given by the formula

$$D = \left\lfloor 255 \ \frac{V - VRL}{VRH - VRL} \right\rfloor$$

[1] It is important that you do not exceed these limits. In this experiment you will be using the variable voltage supply that is available in `PE7` pin. The voltage in this pin will be within the acceptable limits. However, if you try to measure any voltage generated by some external device, it is your responsibility to ensure that you do not exceed these limits.

where

D	The converted digital value
V	Measured voltage
VRL	Lower limit voltage
VRH	Upper limit voltage
$\lfloor \cdots \rfloor$	rounded down value

Note: If the input voltage is below the lower limit, then the converted value would be zero and if the input voltage is above the upper limit, the converted value would be 255. Again, it is not a good idea to go outside the two limits.

9.3 Electrical Connections

1. To measure any analog voltage, you first need to provide the lower and upper limit. These are connected to the HC11 to the pins labelled VRL and VRH. Connect VRL to ground (zero volts) and VRH to five volts. *Double check both these connection before applying power!*

2. Now you can measure the analog voltage connected to any of the `PORT E` pins. Connect the voltage you want to measure to one of the `PORT E` pins. In this experiment, we will measure the voltage available at `PE7`[2] and measure it using pin #3 of `PORT E`. You can now change the voltage using the potentiometer on the board. *Do not use the power supply in the electronics lab as the test voltage. You can accidentally exceed the five volt limit.*

9.4 Decisions, decisions ...

9.4.1 Multiplexing

HC11 provides several options for analog to digital conversions. Every time you perform a conversion, HC11 will actually perform four conversions in sequence, and store the converted values in `ADR1`, `ADR2`,`ADR3`,and `ADR4` at addresses `$1031` to `$1031`. You as a programmer can decide how you want to use the four measurements. You can take four measurements of the same input pin. These measurements will be taken 32 *E-clock* ticks apart or 16 microseconds apart on a standard HC11 running 2 MHz clock. You can expect your for readings to be almost equal. This is the *non-multiplexed* mode. In the *multiplexed* mode, the four measurements are performed on four different `PORT E` pins. Here you can either pick the

[2]This instruction is specifically for FOX11 board. For other boards, connect a 10K potentiometer between five volts and ground. Connect the wiper of the potentiometer to pin #3 of PORT E.

pins PE0-PE3 or the pins PE4-PE7. If you chose the first the group of pins then the voltage in PE0 will be converted and stored in ADR1, the voltage in PE1 will be converted and stored in ADR2, etc. If you chose the second group of pins then the voltage in PE4 will be converted and stored in ADR1, the voltage in PE5 will be converted and stored in ADR2, etc. Note that if you want to measure more than 4 pins then the only way to do it will be to make two sets measurements, one for for PE0-PE3 and the other for PE4-PE7.

9.4.2 Scanning

To take a measurement, you have to initiate the conversion. Once it is initiated, HC11 will take 128 *E-clock* ticks to complete all the four conversion and fill ADR1, ADR2,ADR3,and ADR4. In the *non-scanning* mode, the conversion process stops. In the *scanning* mode, once the converted data is stored in ADR4, HC11 will automatically initiate the next round of conversions to fill ADR1, ADR2,ADR3,and ADR4. In the scanning mode, ADR1, ADR2,ADR3,and ADR4 will always have the most up to date value of the input voltage. However, scanning consumes more power, and can be wasteful if you don't need data at a fast rate.

9.5 Process of taking a measurement

9.5.1 Turning on (powering up) the convetor

Before you can take any measurements, you have to turn on the A/D convertor. The convertor is normally off. You turn it on by setting bit #7 of OPTION which is at location $1039. After power up you need to wait 100 microseconds before you can us the convertor. 100 microseconds is 200 E-clock ticks. You can put in a delay of 100 microseconds with this following do nothing counting loop (40 times around the loop with each pass taking 5 clock cycles):

```
    LDAA #40
LT
    DECA
    BNE LT
```

Note that you need to do this only once in your program.

9.5.2 Initiating a conversion

You initiate a conversion by writing to ADCTL at address $1030. What you write to the location depends on what choices you make. Bit #5 is the SCAN bit. Set

this to 1 if you want scanning; or else, set it to zero. Bit #4 is the `MULT` bit. Set this to 1 if you want multiplexing; or else, set it to zero. Bits #3-#0 specify which `PORT E` pin you want to measure. In the multiplexed mode, you can specify any of the four pins that are part of the multiplexed group.

9.5.3 Making sure you have valid data

Once you have initiated the conversion, you have to make sure you that you have valid data. HC11 will turn on bit #7 of `ADCTL` once all four conversions are complete. It is your responsibility to check this bit before reading the converted values.

9.6 A trial dry run

We will first do some analog to digital conversion without writing any programs.

1. Make the appropriate electrical connections. Make sure that you connect `PE7` to `PE3`, `VRL` to ground and `5V`.

2. First turn on the convertor by storing $80 to location $1039. You do this with the memory modify command: `MM 1039` and entering the value `80`.

3. Since it is going to take lot more than 100 microseconds for you to type anything, there is no need for any waiting for the A/D to warm up. Now we will configure the A/D to be in non-multiplexed, scan mode to measure `PE3`. We do this by entering the bit pattern `00100011/` in location `$1030`. The hexadecimal representation for the bit pattern is `$23`, and we need this for memory modify command. Use the command `MM 1030` and enter the value `23`.

4. Adjust the input voltage so that it is one volt. use the command

   ```
   md 1030 1030
   ```

   and write down the first five numbers. Note that the contents of location `$1030` is `A3` and not `23`. This because HC11 will turn the 7th bit at location `$1030` when the conversion is completed. The next four values should be approximately 255/5 or 51 in decimal or `33` in hex. Verify that the next four values are approximately `33`.

5. Change the input voltage. Do a memory dump to see what the converted values are. Since we configured the analog to digital convertor to be in the

scanning mode, there is no need to issue any new commands. Measure the input voltage, `V` and compute $\frac{255V}{5}$. Also note down the values in locations `$1031-$1034`. Fill in the following table for different input voltages and verify your results.

Measured Voltage (V)	$\left\lfloor 255\left(\frac{V}{5}\right)\right\rfloor$ In Decimal	$\left\lfloor 255\left(\frac{V}{5}\right)\right\rfloor$ In Hex	Memory dump			
			$1031	$1032	$103	$1034

9.7 A simple digital voltmeter

We can write a simple voltmeter program to measure voltages in the zero to 5 volts range. Since the output is intended for human eyes, we want the output to be in decimal. We will use one decimal accuracy so that the numbers will be between 0 and 50 (decivolts) and we will put a decimal point between the two digits. To convert the digital values to decivolts, we need to multiply by 50 and then divide by 255. The steps are as follows:

1. Turn the A/D on and wait the required 100 microseconds

```
    LDAA #%10000000
    STAA OPTION

    LDAA #40
LT
    DECA
    BNE LT
```

2. Initiate conversion with `SCAN=1`, `MULT=0`, `PIN=3`

```
    LDAA #%00100011 *SCAN=1, MULT=0, PIN = 011 (PE3)
    STAA ADCTL
```

3. Make sure that the conversion is complete

```
WCCF
    LDAA ADCTL
    ANDA #%10000000
    BEQ  WCCF
```

4. Take a reading (this is easy!)

```
LDAA ADR1
```

5. Convert it to decivolts by multiplying by 50 and then dividing by 255

```
LDAB #50
MUL
LDX #255
IDIV
```

The quotient as a result of division is in the X register. This is an awkward destination as most of the functions expect the value on the `A` register. We transfer from X to A in two steps.. first from X to D. Now the result is in the [A—B] pair and since the value is less than 255 (the value is between 0 and 50), the result is in the B register. We transfer the value from B to A.

```
XGDX * D <-> X
TBA  * B -> A
```

6. Print the value in decimal We can now convert the data to BCD format and then print the left and the right digit.

```
JSR HEX2BCD

PSHA    *SAVE IT FOR LATER USE
JSR OUTLHLF

LDAA #'.
JSR OUTA

PULA    *GET THE SAVED VALUE
JSR OUTRHLF

JSR OUTCRLF
```

The code to convert to BCD was discussed in an earlier section and is produced here for reference:

```
HEX2BCD
    PSHB
```

```
        TAB          *COPY A TO B
    H2B.LT
        CMPB #10     *IS B < 10
        BLO  H2B.DONE    *IF SO WE ARE DONE

        SUBB #10     *B <- B-10
        ADDA #6      *A <- A+6
        BRA H2B.LT

    H2B.DONE
        PULB
        RTS
```

7. We can go back and repeat the steps (the first need not be performed more than once). It is convenient to check the keyboard and go back only if the user has not pressed the ESC key (code 27).

The complete program to achieve these steps is given below

```
;;;;;;;;;;;;;;;;;;;;Start: A/D code ;;;;;;;;;;;;;;;;;;;;
OUTLHLF EQU $FFB2
OUTRHLF EQU $FFB5
OUTCRLF EQU $FFC4
INPUT   EQU $FFAC
OUTA    EQU $FFB8

OPTION  EQU     $1039
ADCTL   EQU     $1030
ADR1    EQU $1031

    ORG $C000


;STEP 1
;TURN A/D ON
    LDAA #%10000000
    STAA OPTION

    LDAA #40
LT
    DECA
    BNE LT
```

```
;STEP 2
; INITIATE CONVERSION
;
    LDAA #%00100011 *SCAN=1, MULT=0, PIN = 011 (PE3)
    STAA ADCTL


;MAIN LOOP
LOOP

;STEP 3
; WAIT FOR CCF

WCCF
    LDAA ADCTL
    ANDA #%10000000
    BEQ  WCCF

;STEP 4
; GET THE CONVERTED VALUE

    LDAA ADR1 *ANY ADRx IS OK

;STEP 5
; CONVERT TO DECIVOLTS

    LDAB #50
    MUL
    LDX #255
    IDIV

; GET THE RESULT IN A REGISTER

    XGDX * D <-> X
    TBA  * B -> A

;PRINTIT
    JSR HEX2BCD

    PSHA    *SAVE IT FOR LATER USE
```

```
    JSR OUTLHLF

    LDAA #'.
    JSR OUTA

    PULA    *GET THE SAVED VALUE
    JSR OUTRHLF

    JSR OUTCRLF

; BACK FOR MORE UNLESS USER TYPES AN ESCAPE
    JSR INPUT
    CMPA #27
    BNE LOOP

    SWI



;FUNCTION TO CONVERT HEX TO BCD
;

HEX2BCD
    PSHB
        TAB          *COPY A TO B
H2B.LT
        CMPB #10     *IS B < 10
        BLO  H2B.DONE    *IF SO WE ARE DONE

        SUBB #10     *B <- B-10
        ADDA #6      *A <- A+6
        BRA H2B.LT

H2B.DONE
        PULB
    RTS
;;;;;;;;;;;;;;;;;;;;End: A/D code ;;;;;;;;;;;;;;;;;;;;
```